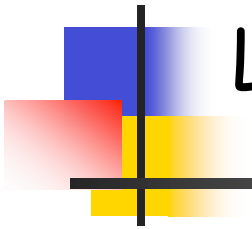
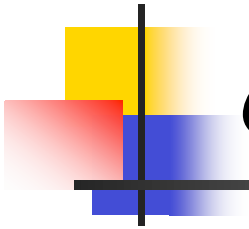


Using Functional Independence Conditions to Optimize the Performance of Latency-Insensitive Systems

Cheng-Hong Li and Luca Carloni

Department of Computer Science
Columbia University in the City of New York

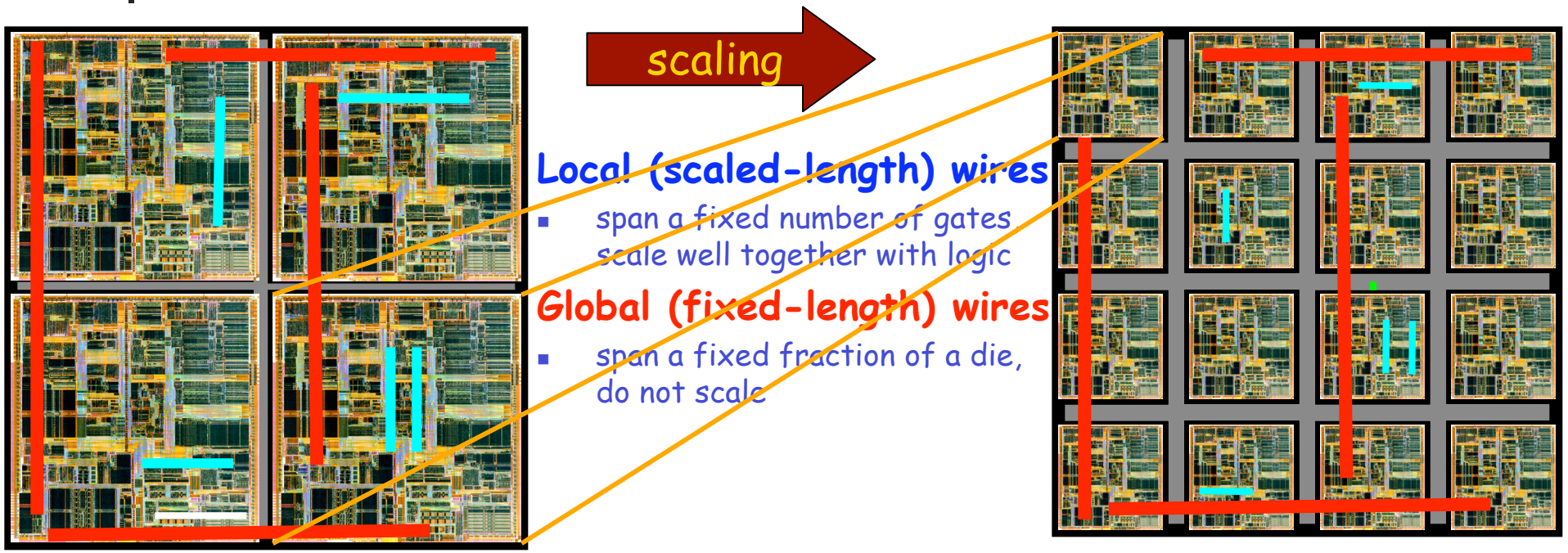
ICCAD 2007 • San Jose • Nov. 6, 2007



Contributions

- Throughput optimization using Functional Independence Conditions (FICs)
  - new circuit template for shell interfaces (FIC-shells)
  - algorithm to synthesize FIC-detect logic
  - effectiveness of FIC proven on a real design
    - FIC occurs frequently in all cases
    - little area overhead
    - can bring throughput back to 1 (the ideal) in certain cases

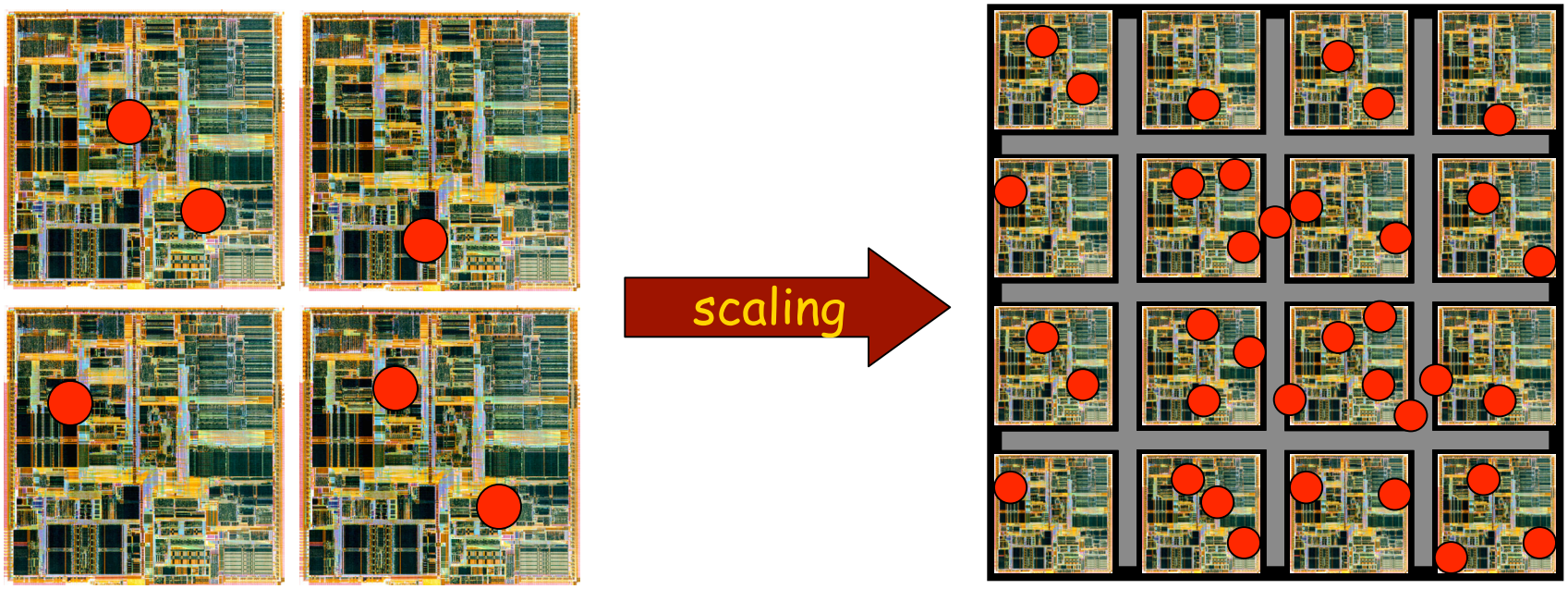
Motivation: With Nanometer Technologies Chips are Becoming Distributed Systems



- Interconnect Latency
  - hard to estimate because affected by many phenomena
    - process variations, cross-talk, power-supply drop variations
  - breaks the synchronous assumption
    - that lies at the basis of design automation tool flows

Motivation: Design Exceptions

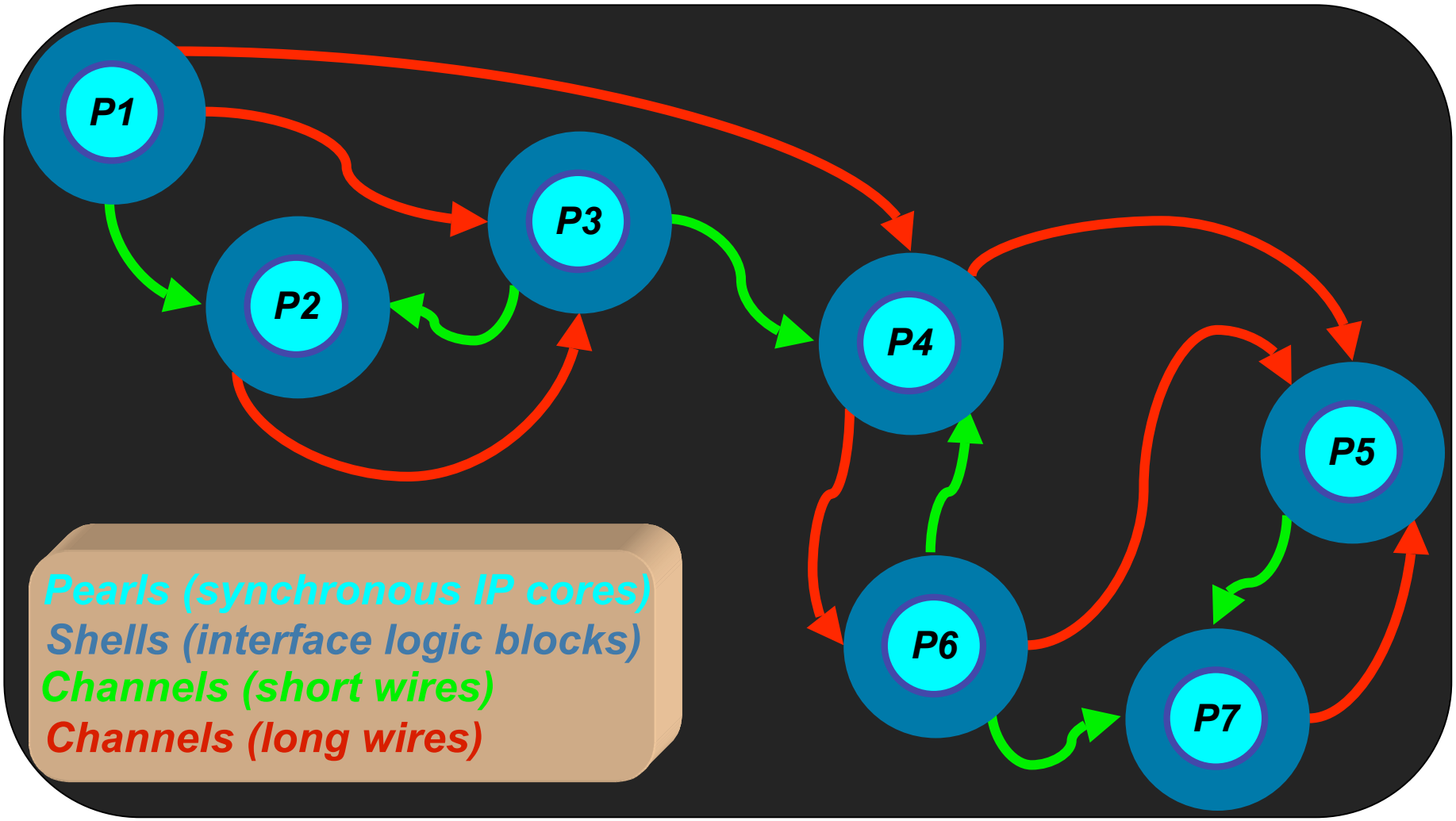
- As technology scales, long wire delays increase the number of design exceptions after physical design of a synchronous system *[Ho et al. 2001]*



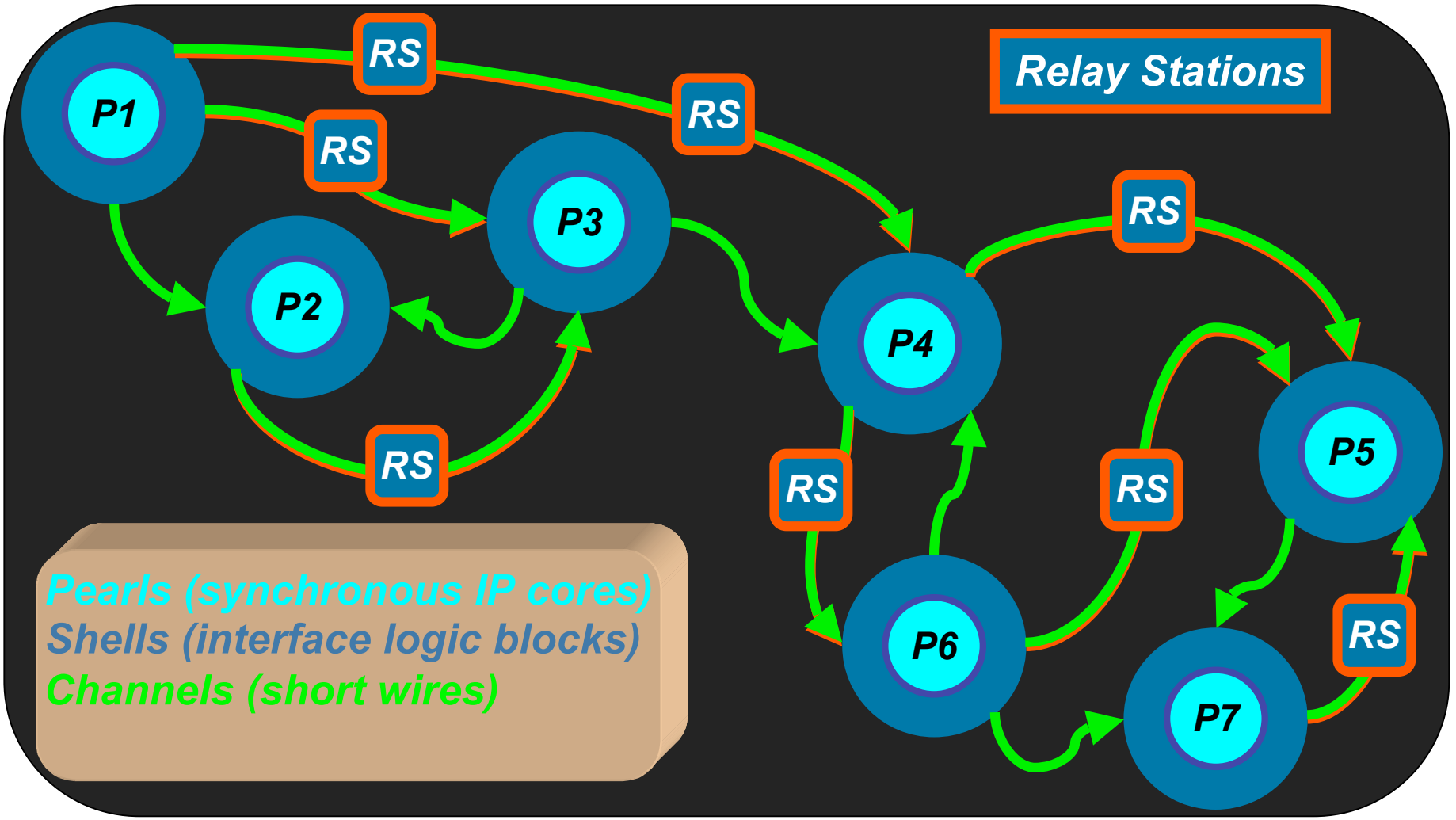
4 blocks, 6 exceptions

16 blocks, 36 exceptions

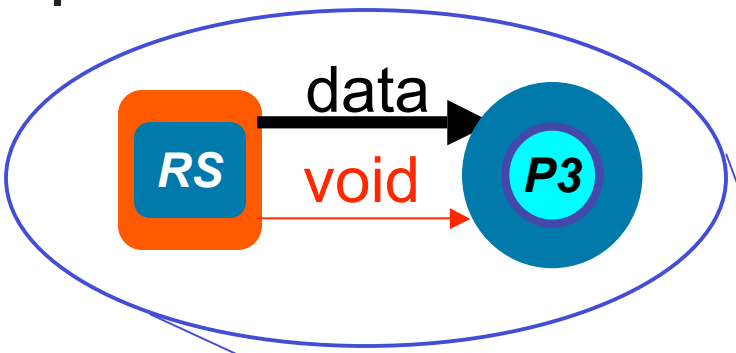
Latency Insensitive Design



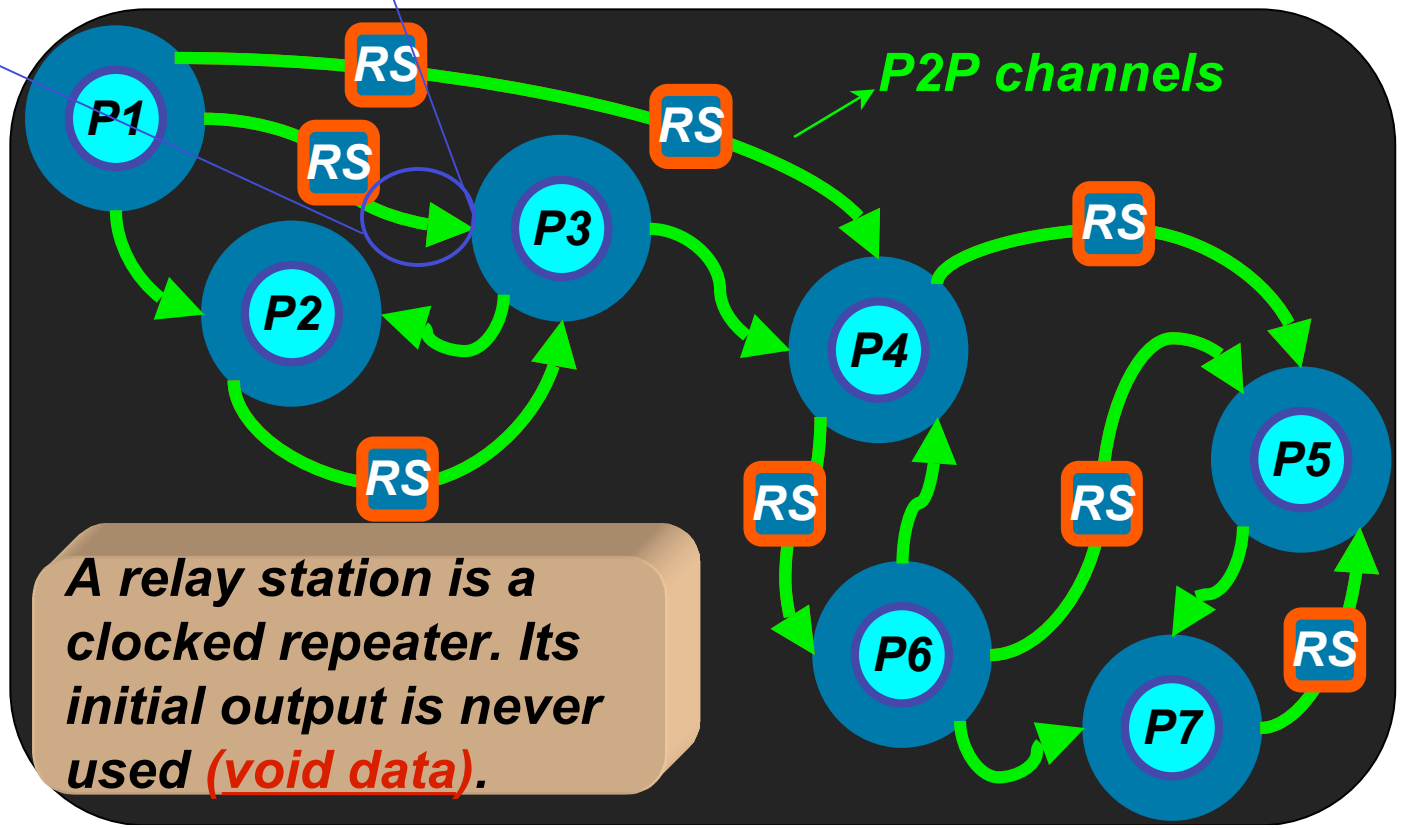
Channel Segmentation (Wire Pipelining)



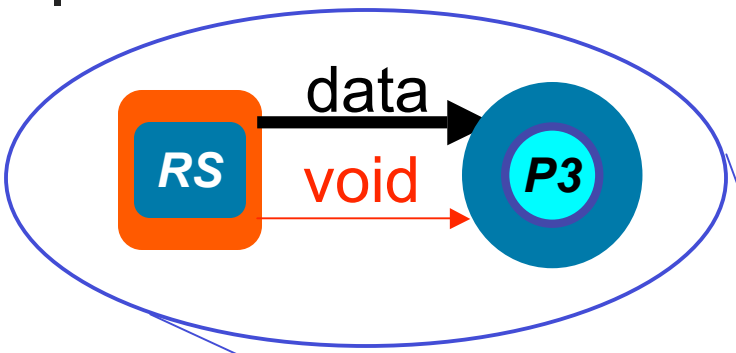
Latency-Insensitive Protocol



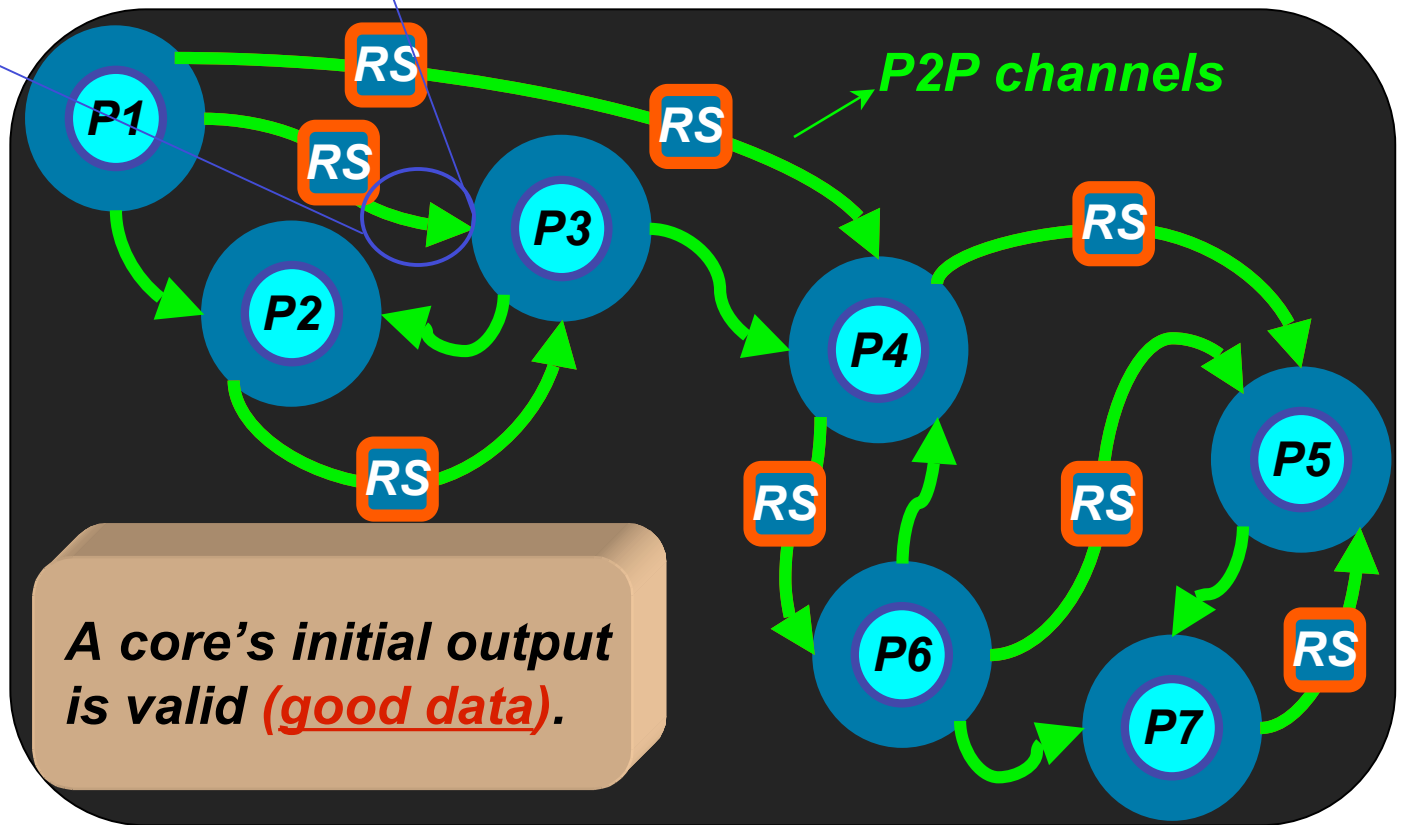
Each channel is augmented with a single bit signal ***void***, indicating the data validity



Latency-Insensitive Protocol

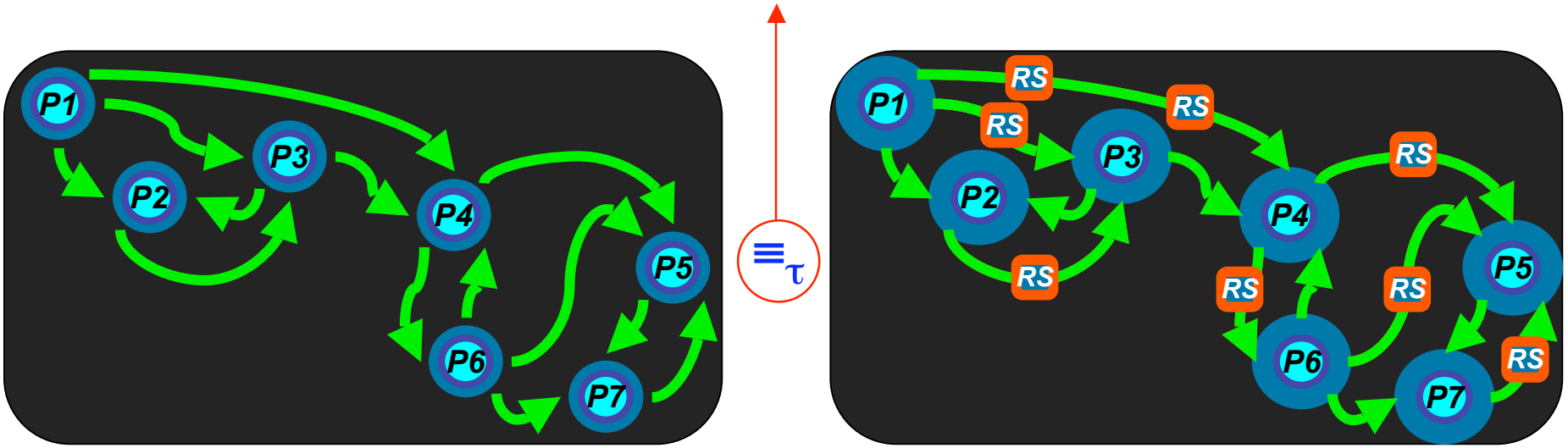


Each channel is augmented with a single bit signal ***void***, indicating the data validity



Equivalence of System Behaviors

the behaviors of these two systems are equivalent



original (strict) system

latency-insensitive

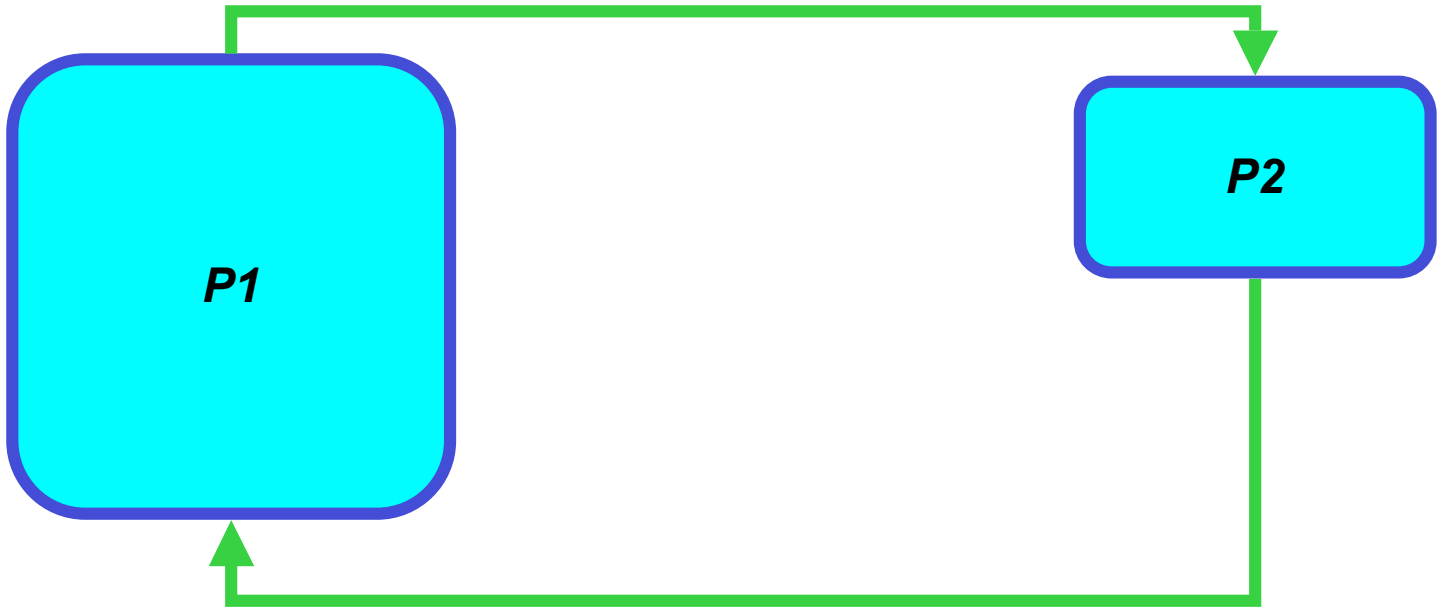
- Latency equivalence: two systems produce the same sequence of valid data tokens
  - possibly with different numbers of interleaved void tokens

System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

  over time observed at system output

Example: RS in a cycle

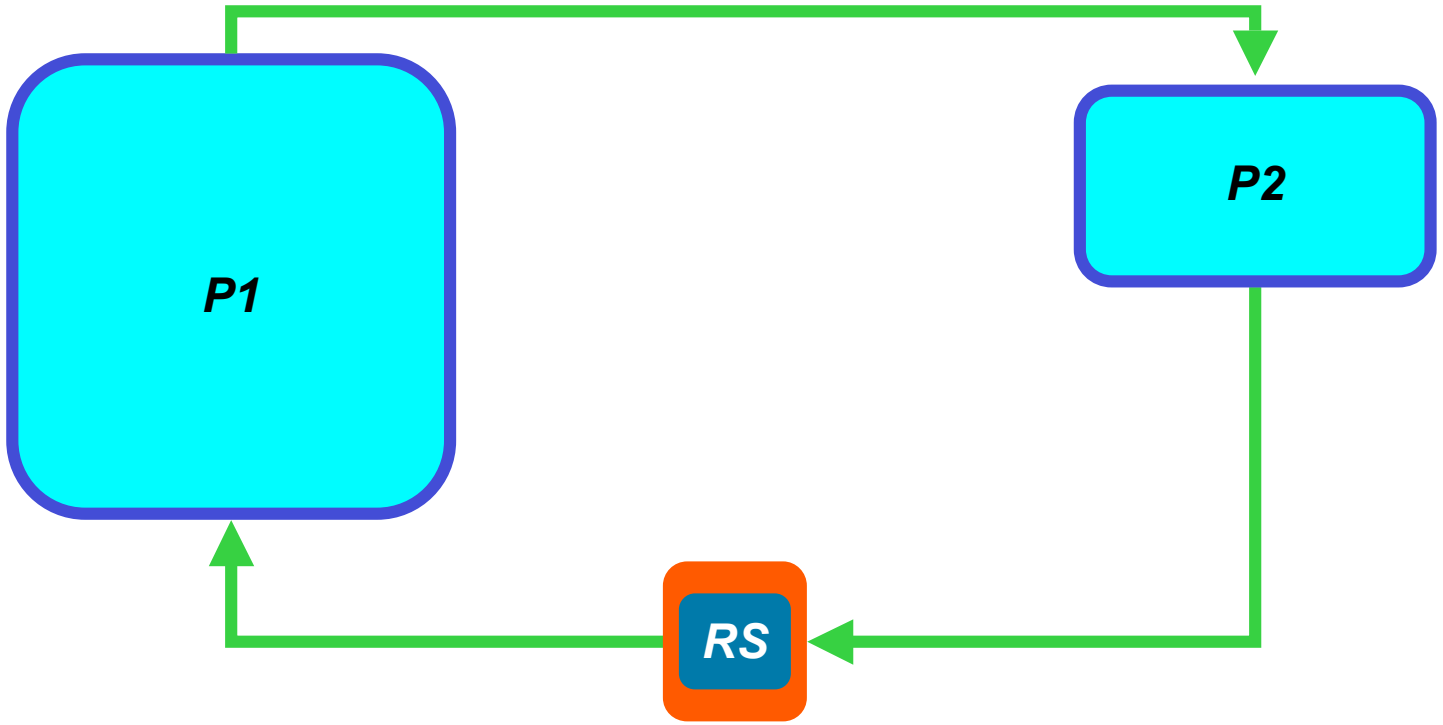


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

  over time observed at system output

Example: RS in a cycle

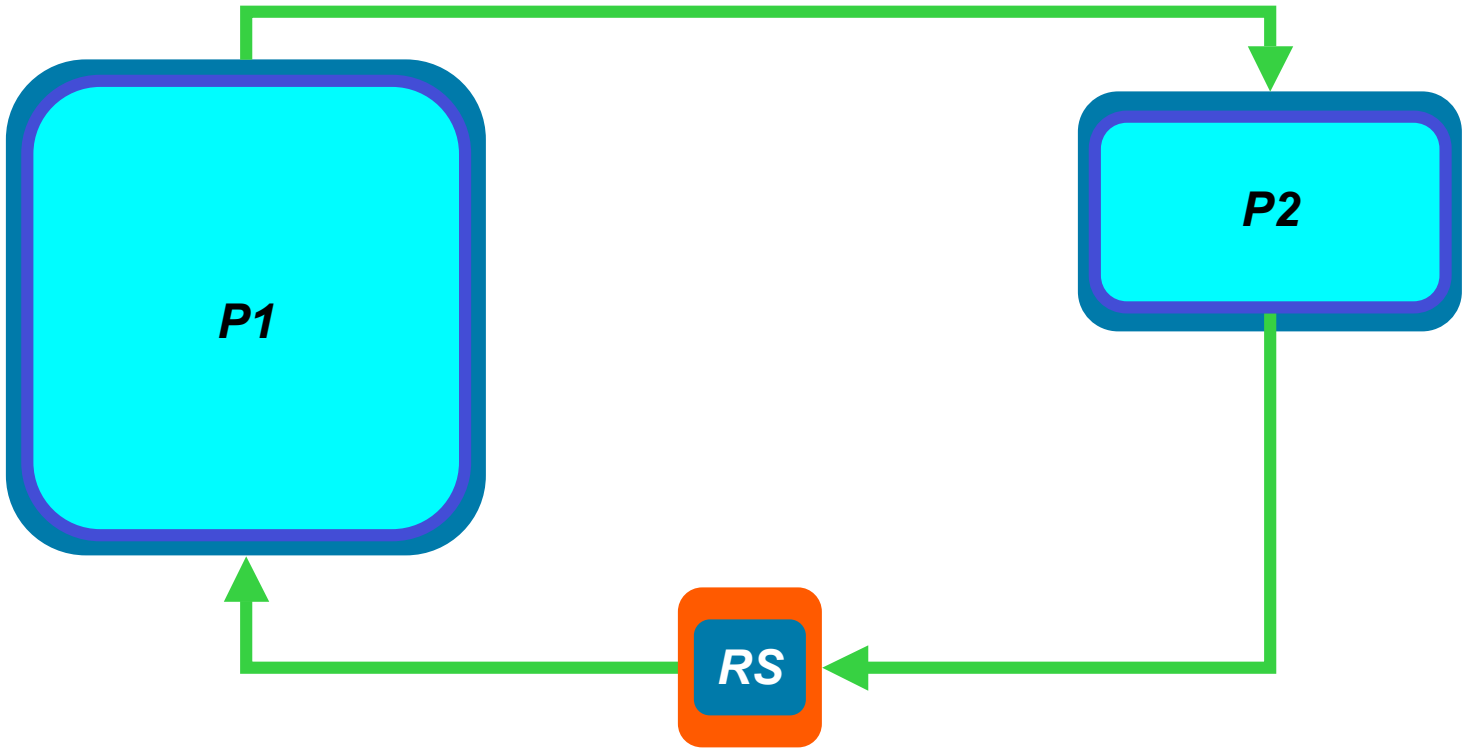


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

  over time observed at system output

Example: RS in a cycle

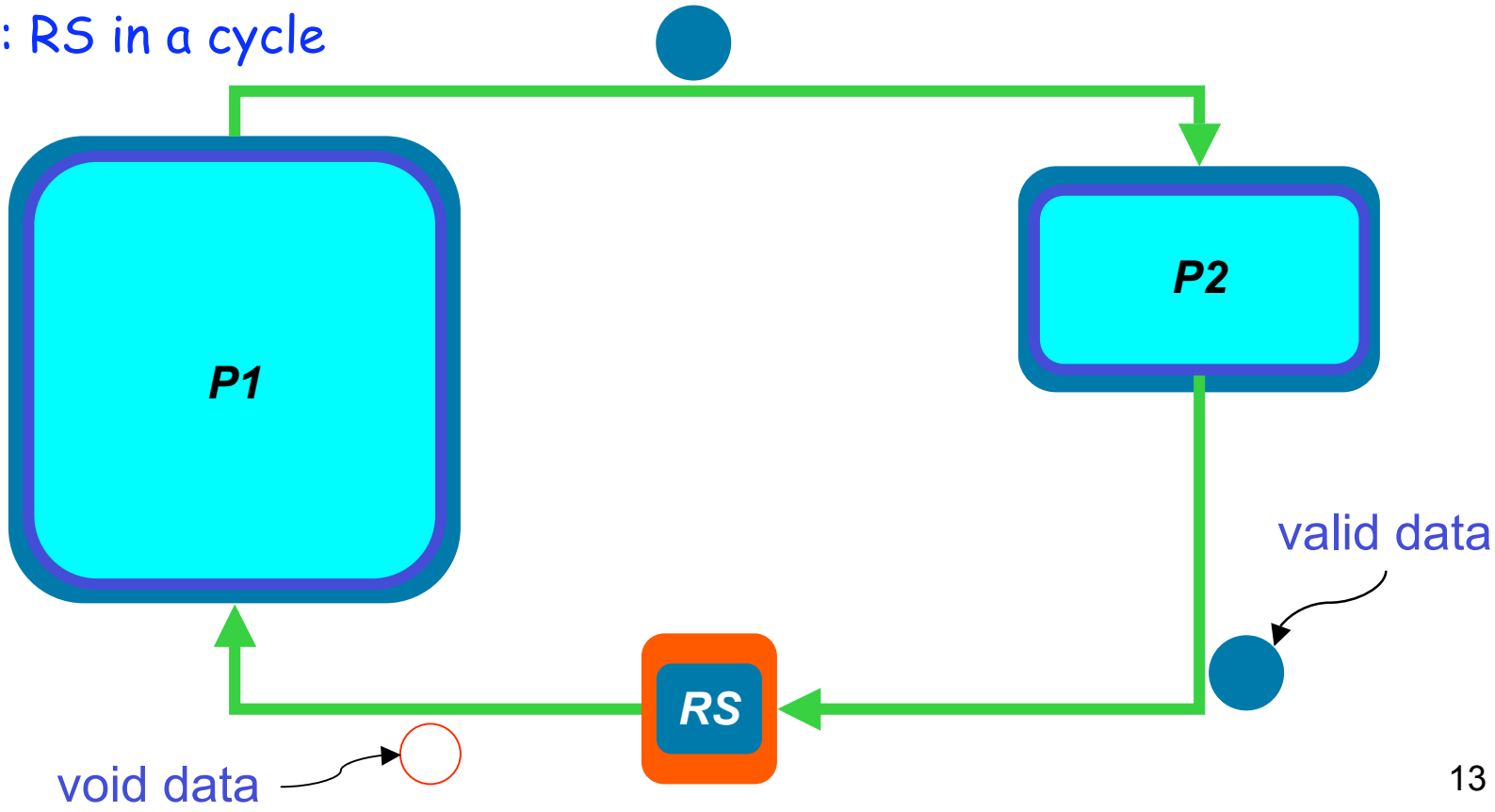


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

over time observed at system output

Example: RS in a cycle

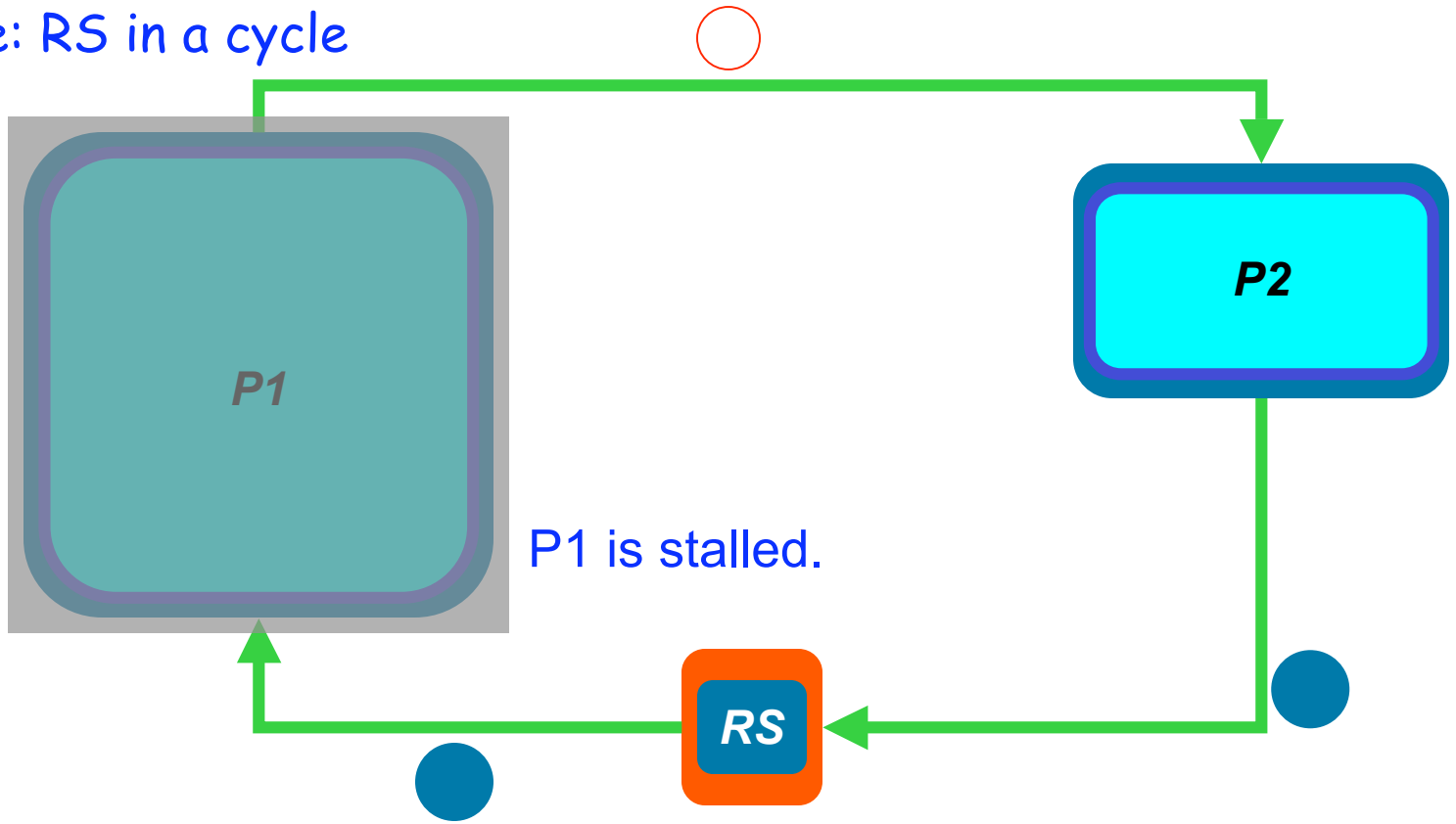


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

  over time observed at system output

Example: RS in a cycle

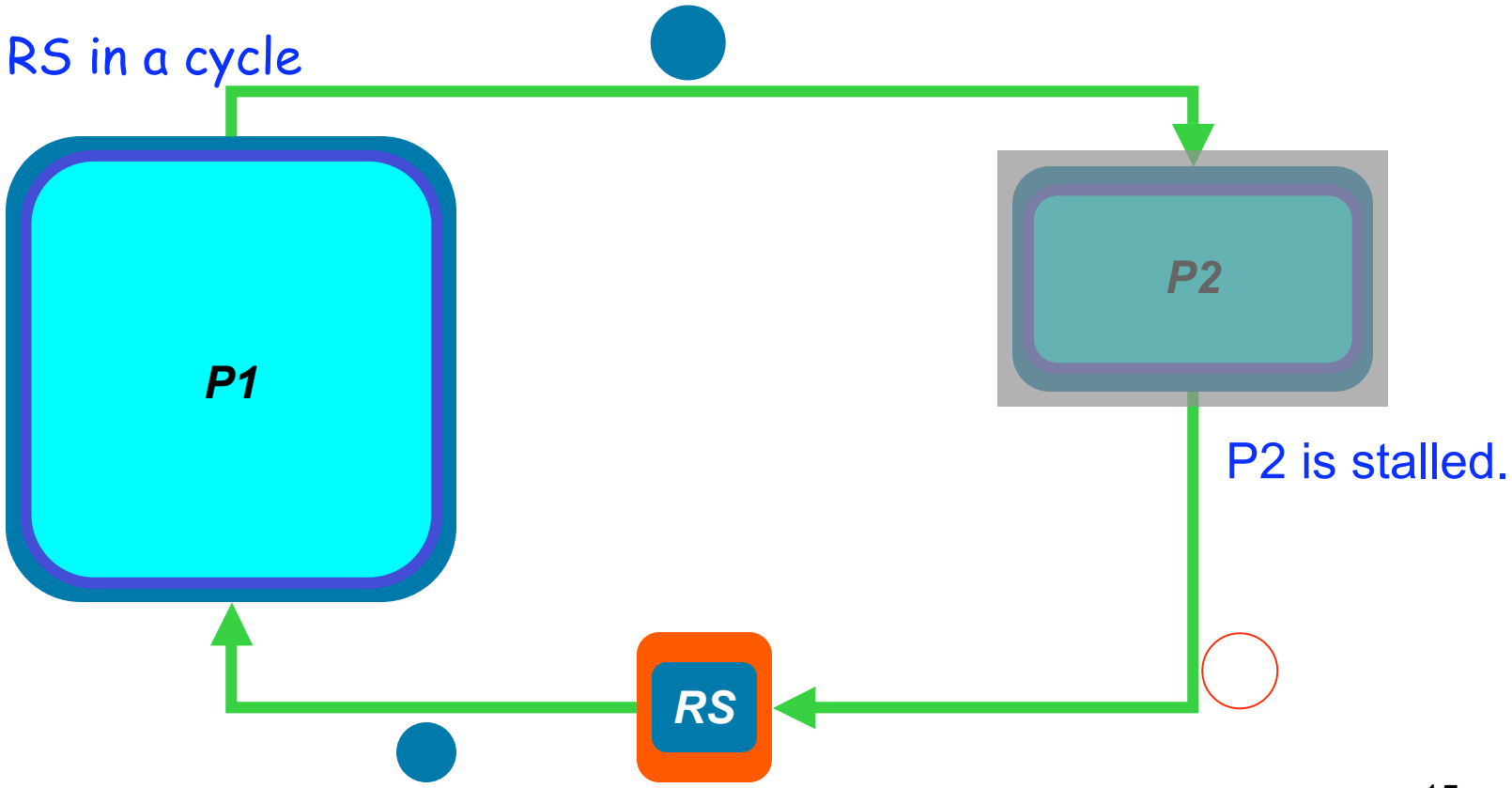


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

  over time observed at system output

Example: RS in a cycle

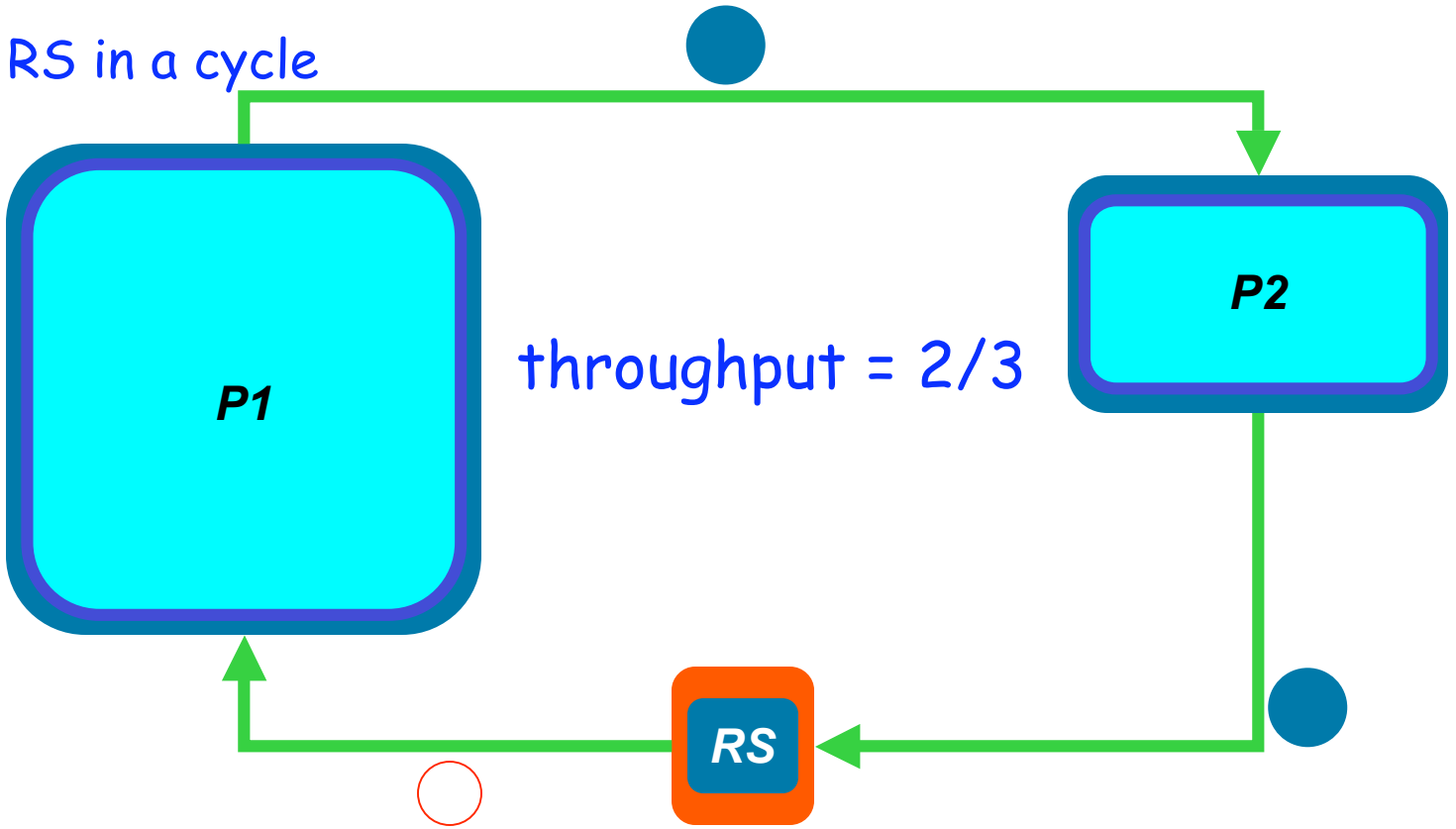


System Throughput Analysis

- Definition: $\frac{\text{(\# of good tokens)}}{\text{(\# all tokens generated)}}$

over time observed at system output

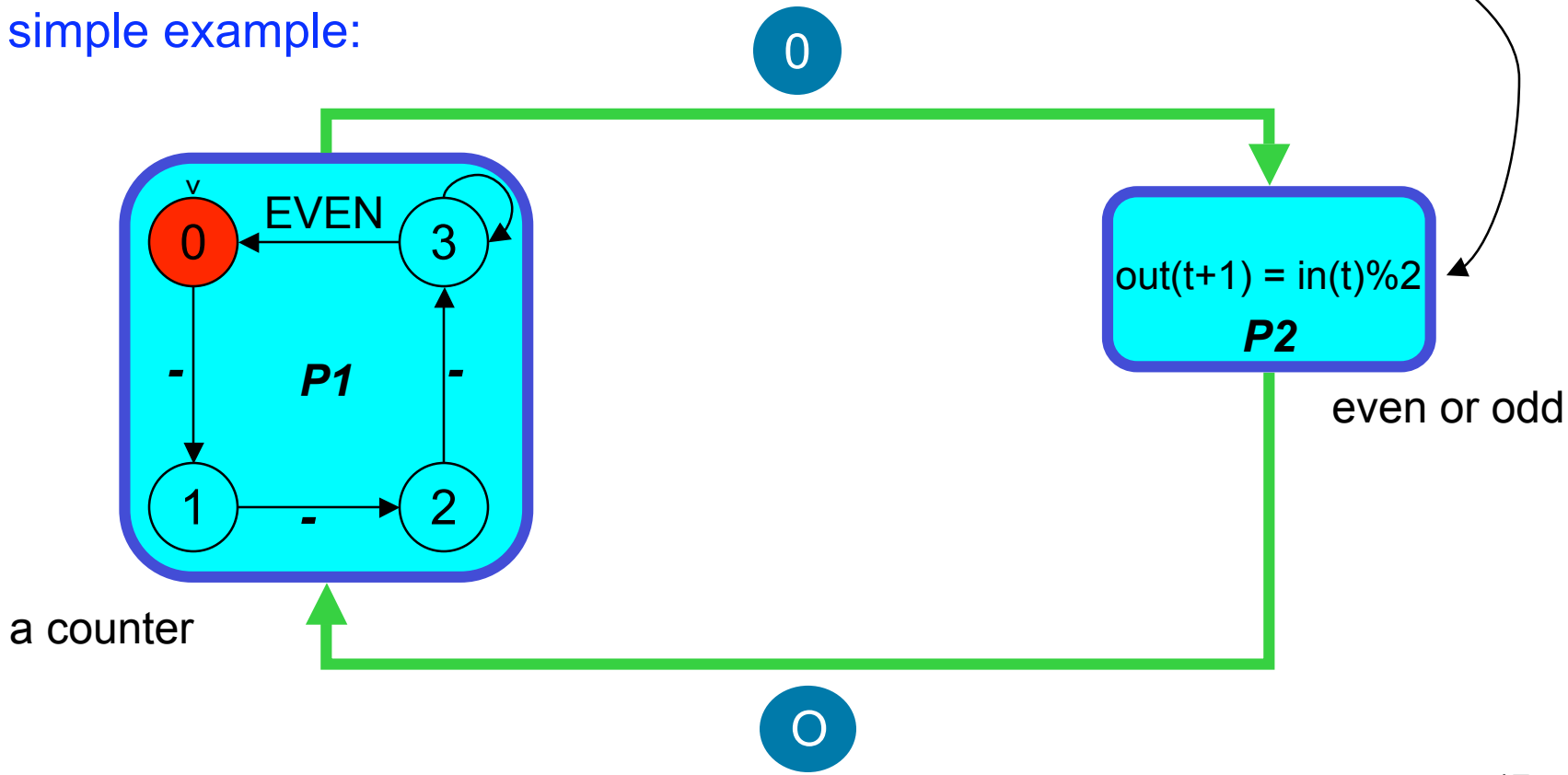
Example: RS in a cycle



Throughput Optimization Using Functional Independence Conditions

P2's output will be *OEOEOE*.....

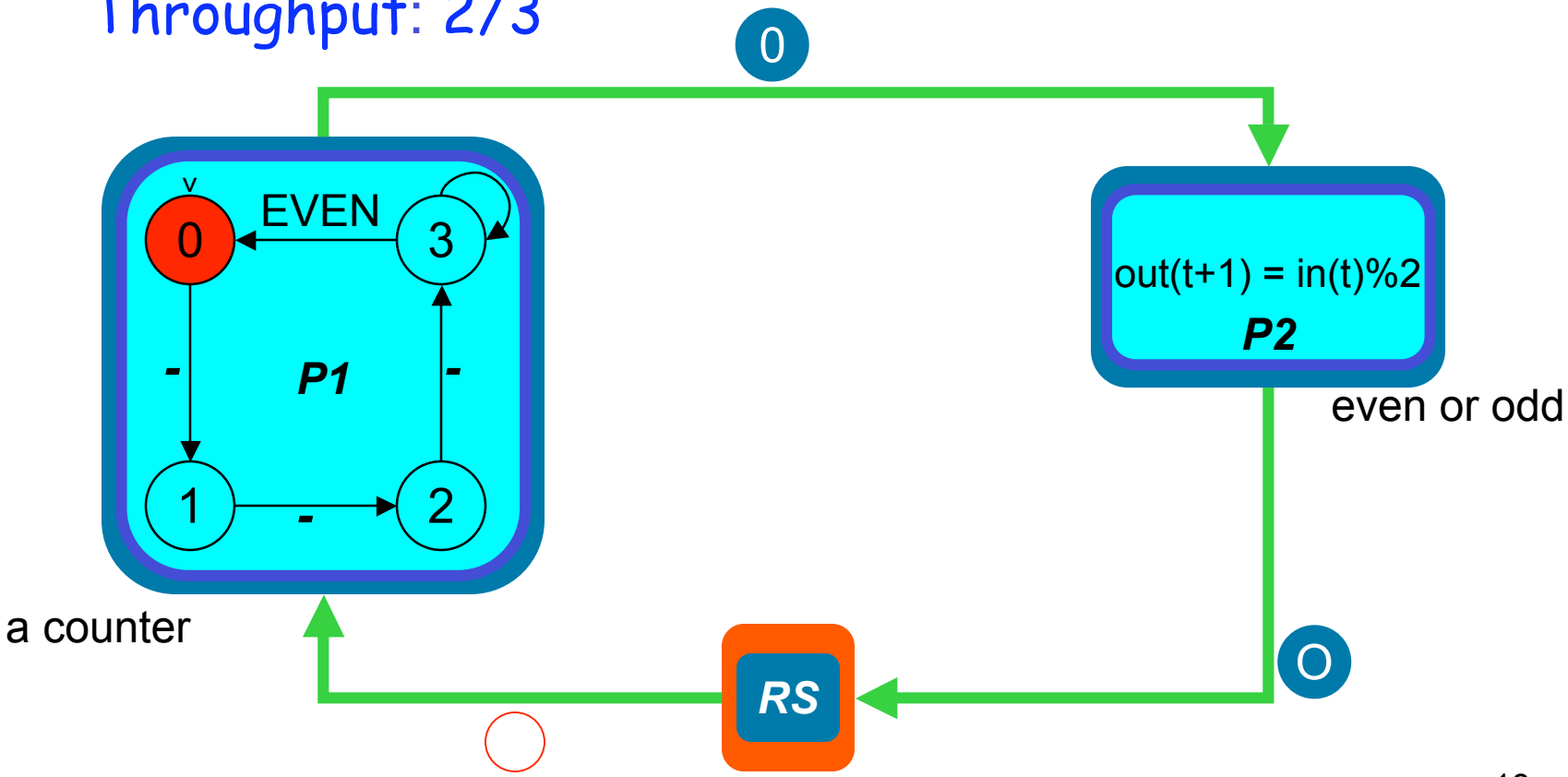
A simple example:



Throughput Optimization Using Functional Independence Conditions

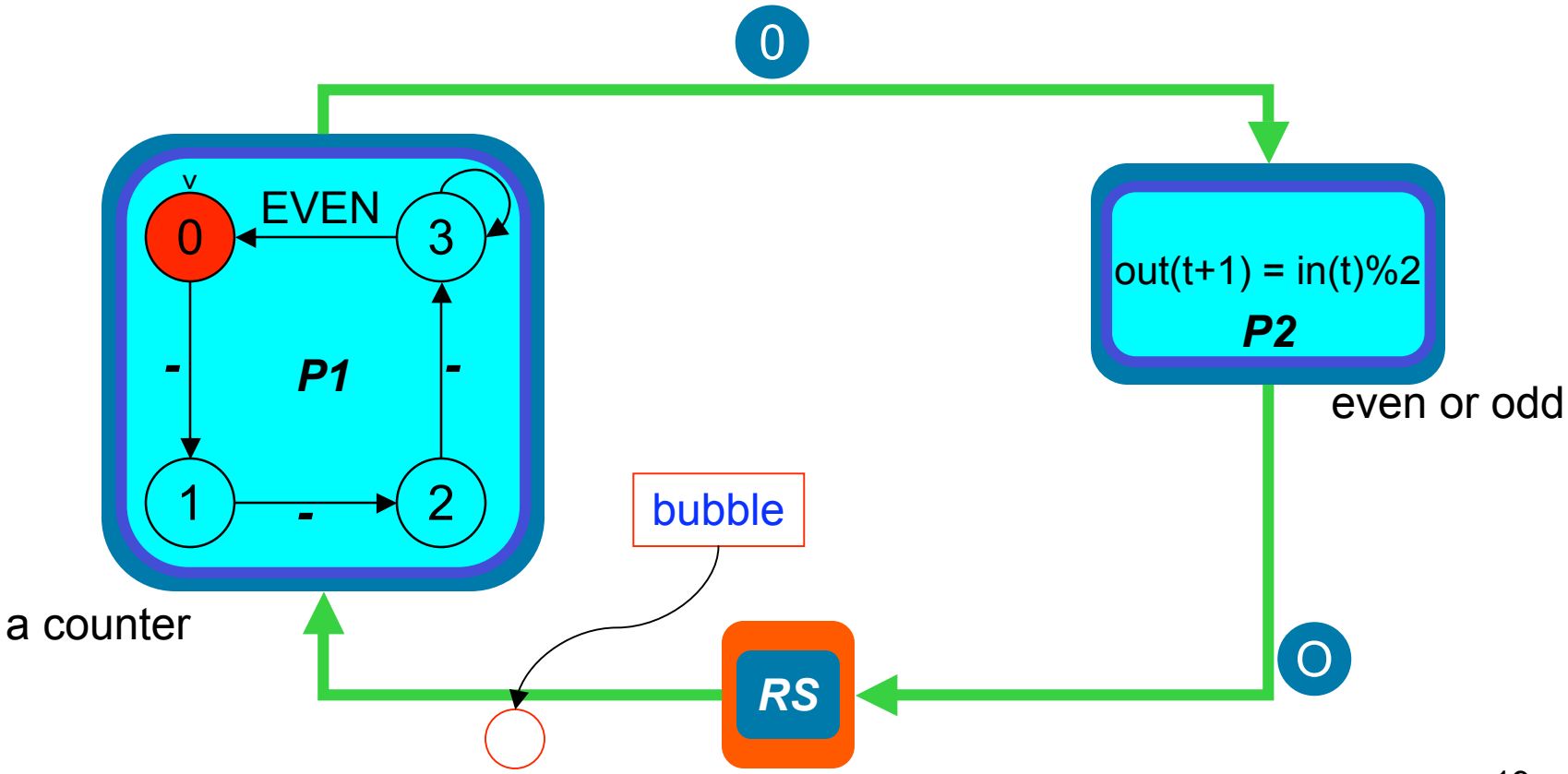
- After applying latency-insensitive design, with one RS
  - P2's trace: *OE#OE#OE#* …
  - Reference: *OEOEOE..*

Throughput: 2/3



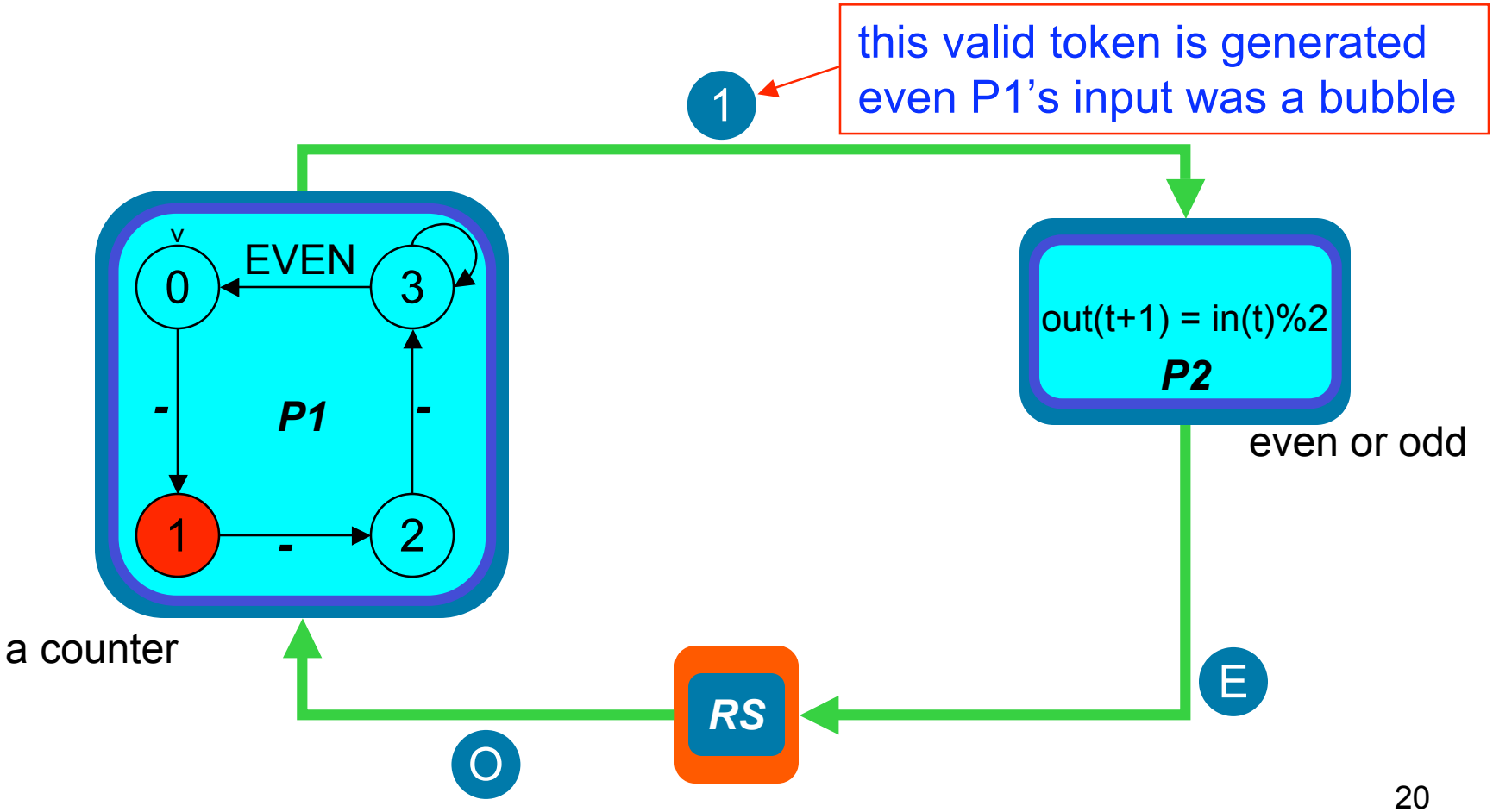
Throughput Optimization Using Functional Independence Conditions

- By exploiting the internal design of cores ("white boxes"), some stalls can be avoided



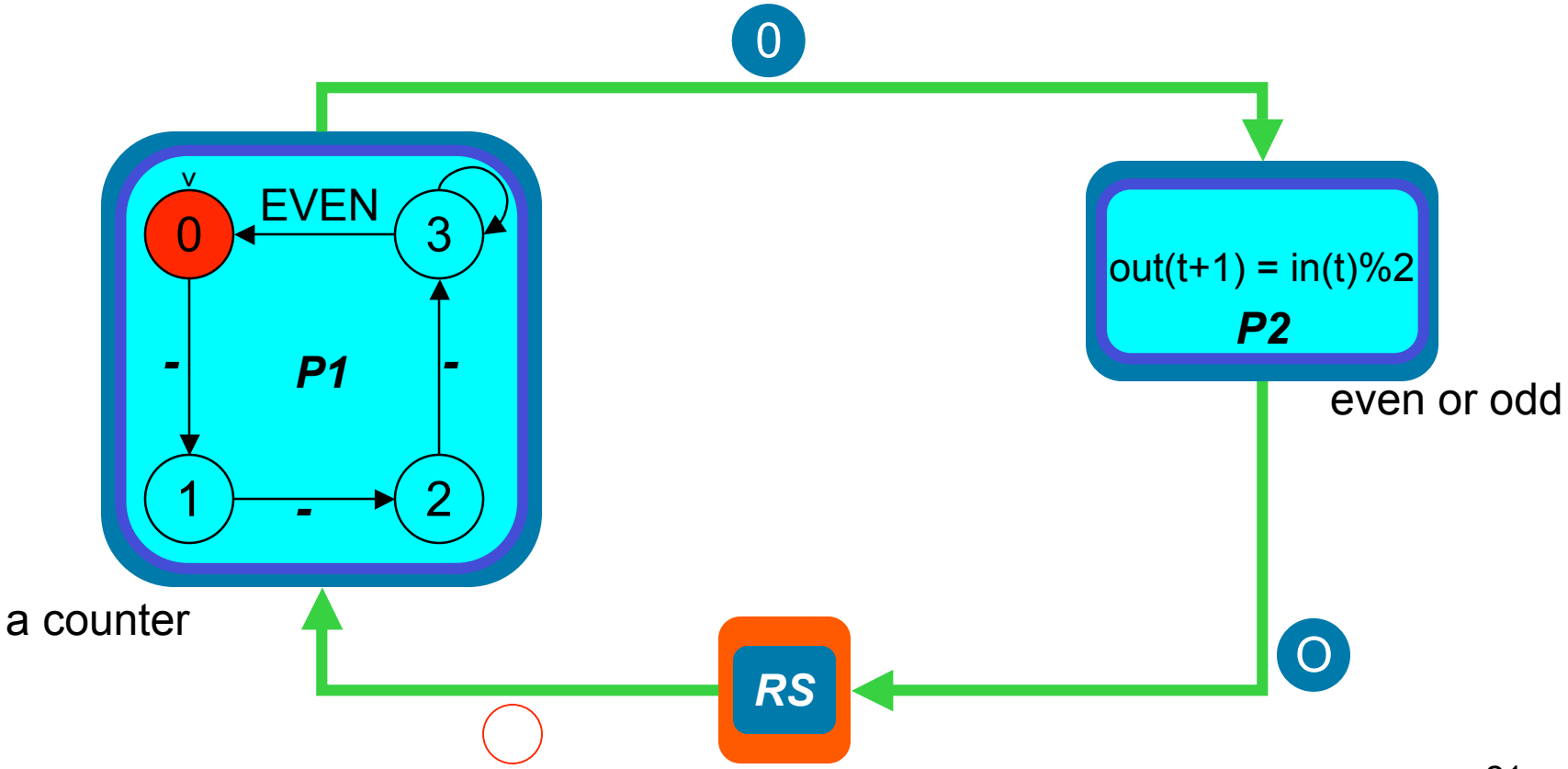
Throughput Optimization Using Functional Independence Conditions

- Under functional independence conditions (FICs), an input is not needed for computation



Throughput Optimization Using Functional Independence Conditions

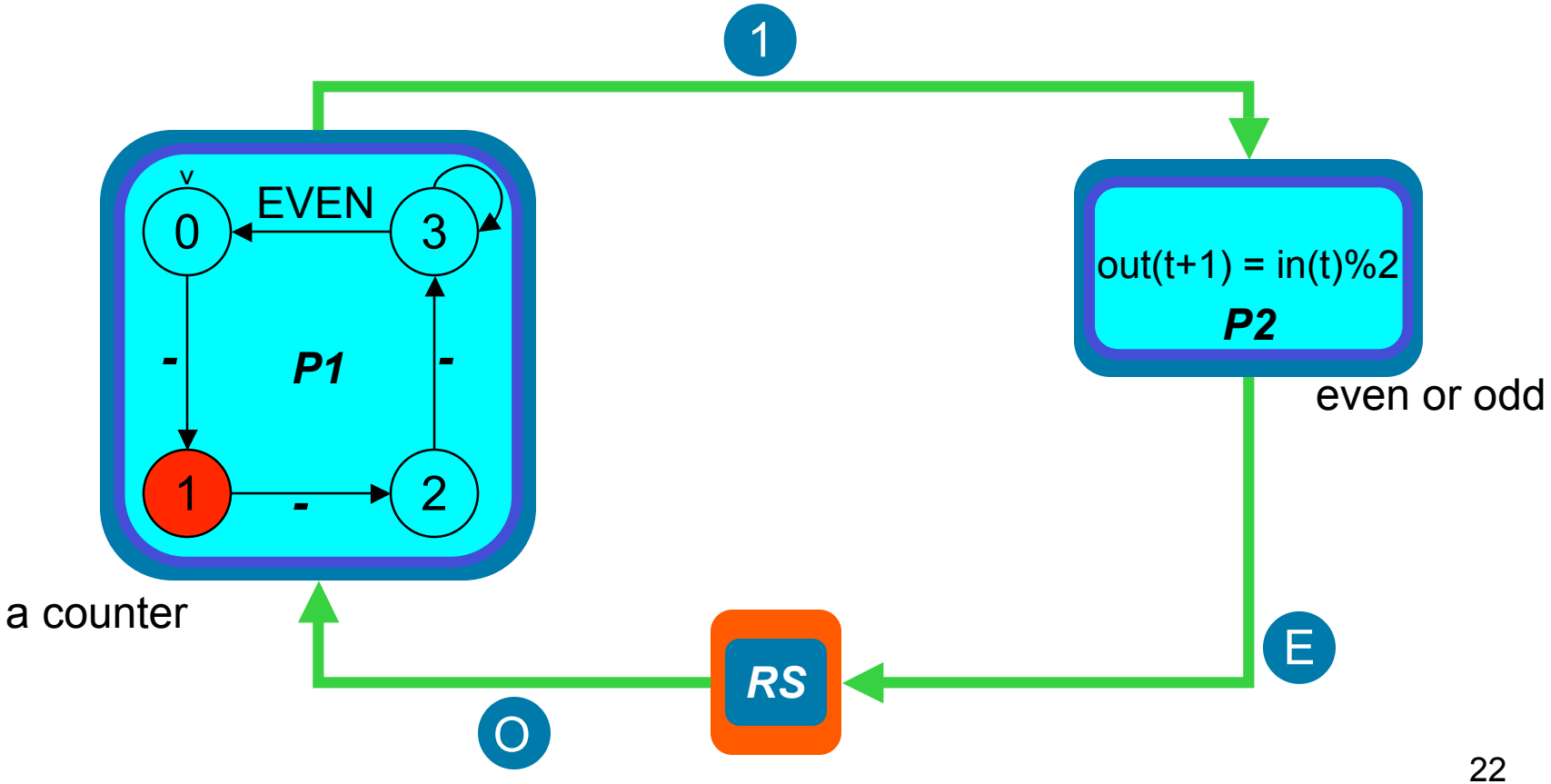
- Cycle 1
  - P2's trace: *O*
  - Reference: *OEOEOEOE…*



Throughput Optimization Using Functional Independence Conditions

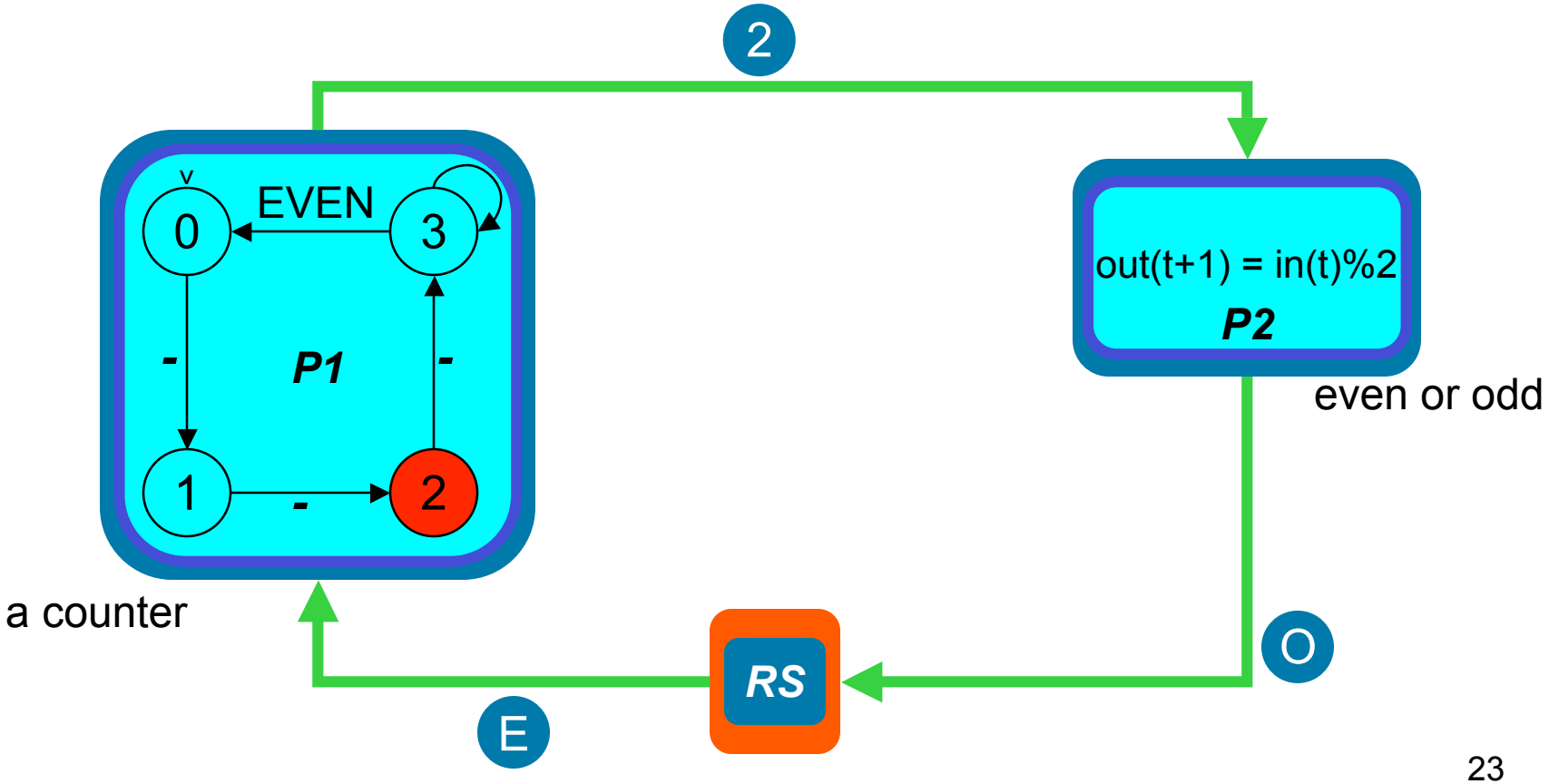
- Cycle 2:
  - P2's trace: *OE*
  - Reference: *OEOEOEOE…*

Bubble was absorbed by P1's firing.



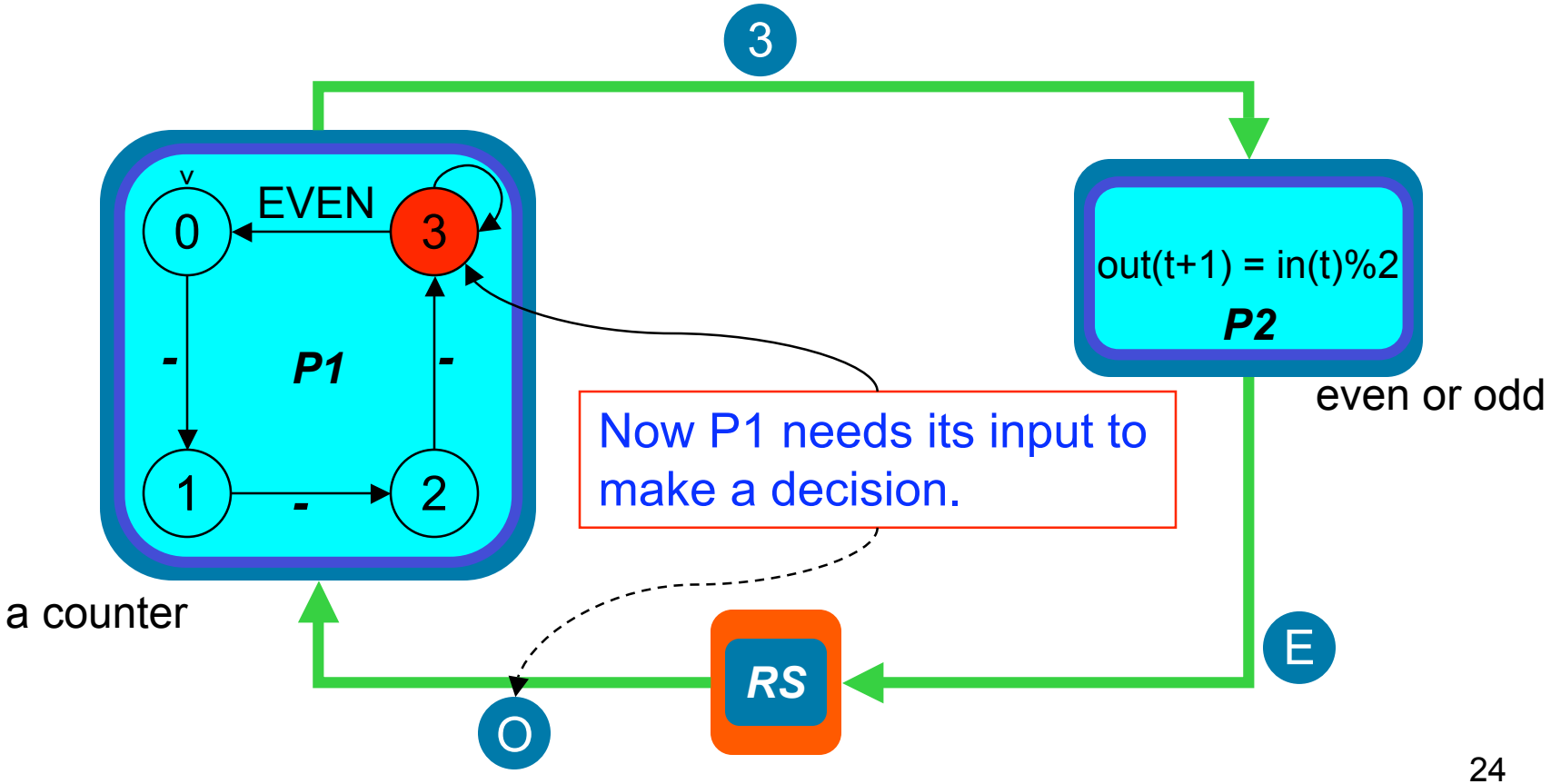
Throughput Optimization Using Functional Independence Conditions

- Cycle 3:
  - P2's trace: *OEO*
  - Reference: *OEOEOEOE…*



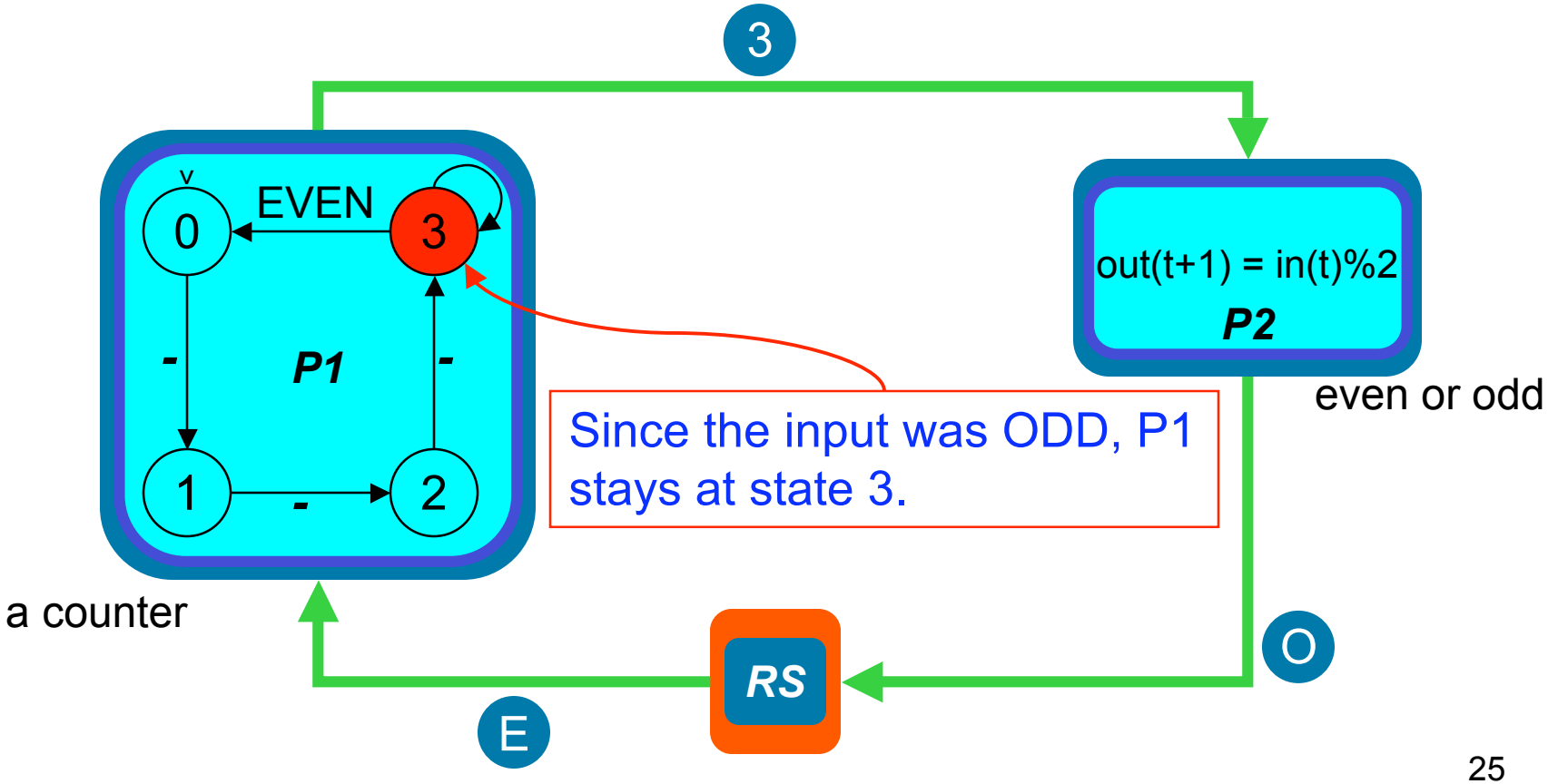
Throughput Optimization Using Functional Independence Conditions

- Cycle 4:
  - P2's trace: *OEOE*
  - Reference: *OEOEOEOE…*



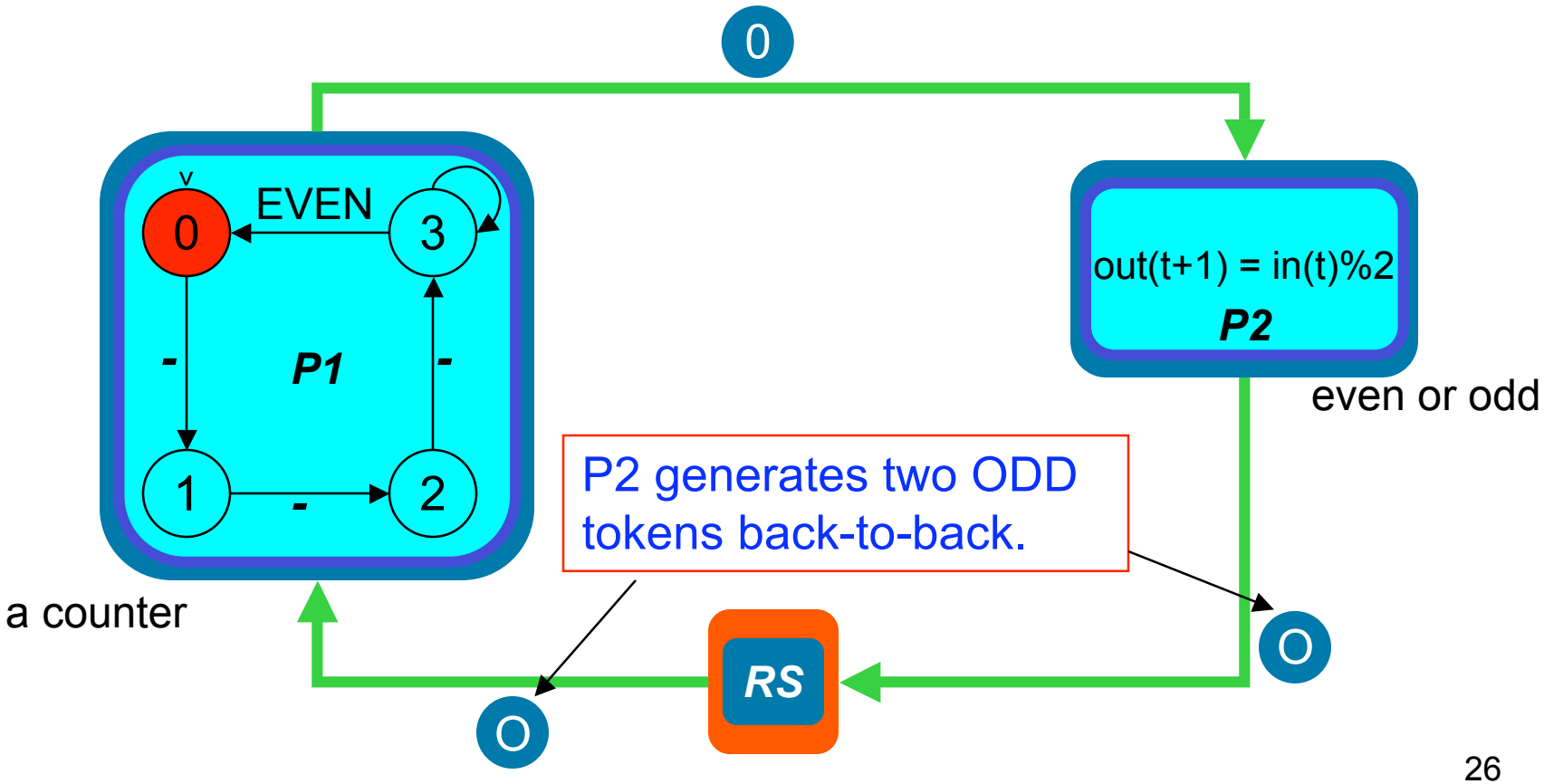
Throughput Optimization Using Functional Independence Conditions

- Cycle 5:
  - P2's trace: *OEOEO*
  - Reference: *OEOEOEOE…*



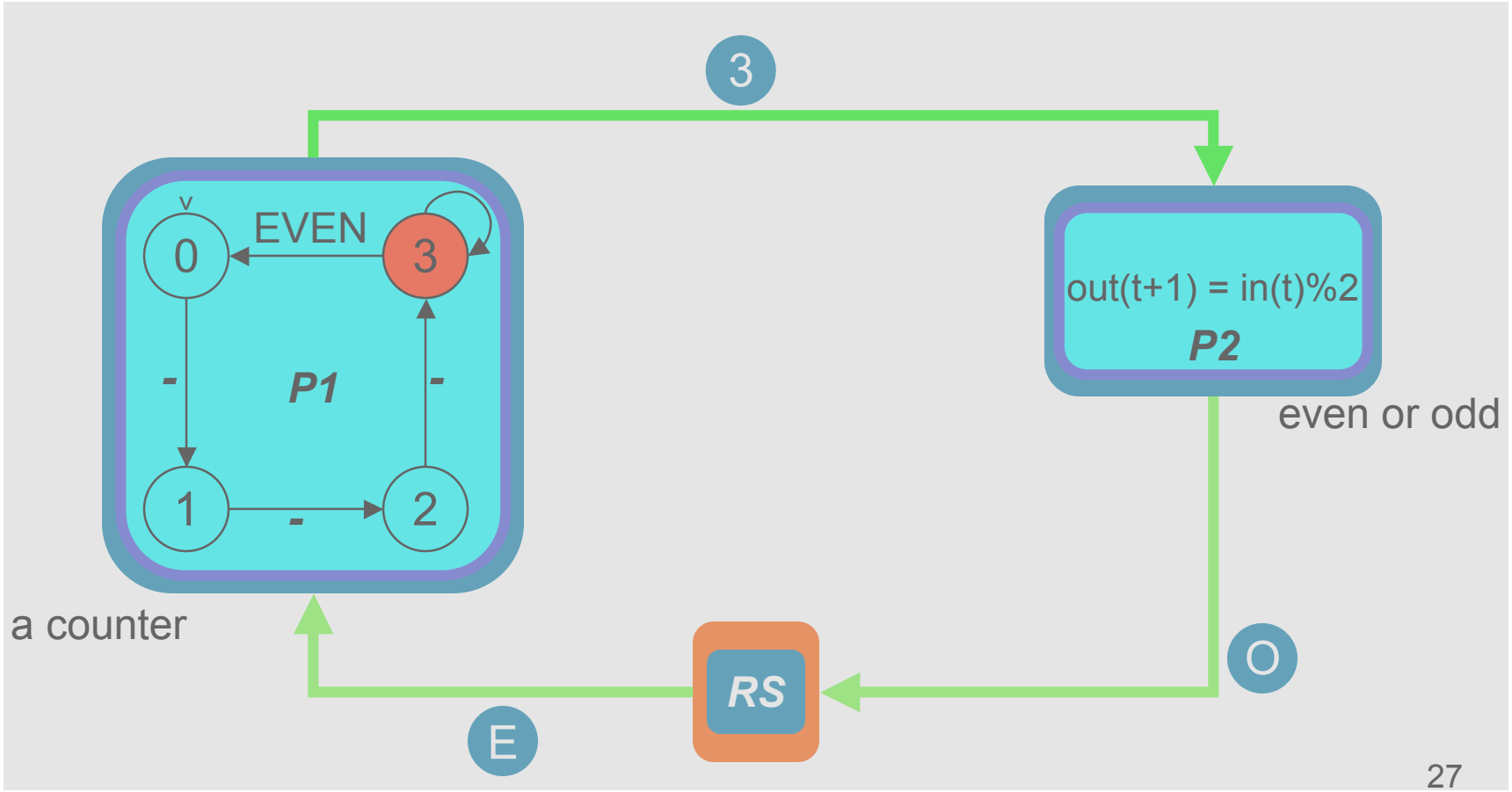
Throughput Optimization Using Functional Independence Conditions

- Cycle 6:
  - P2's trace: *OEOEOO -> WRONG!!*
  - Reference: *OEOEOEOE…*



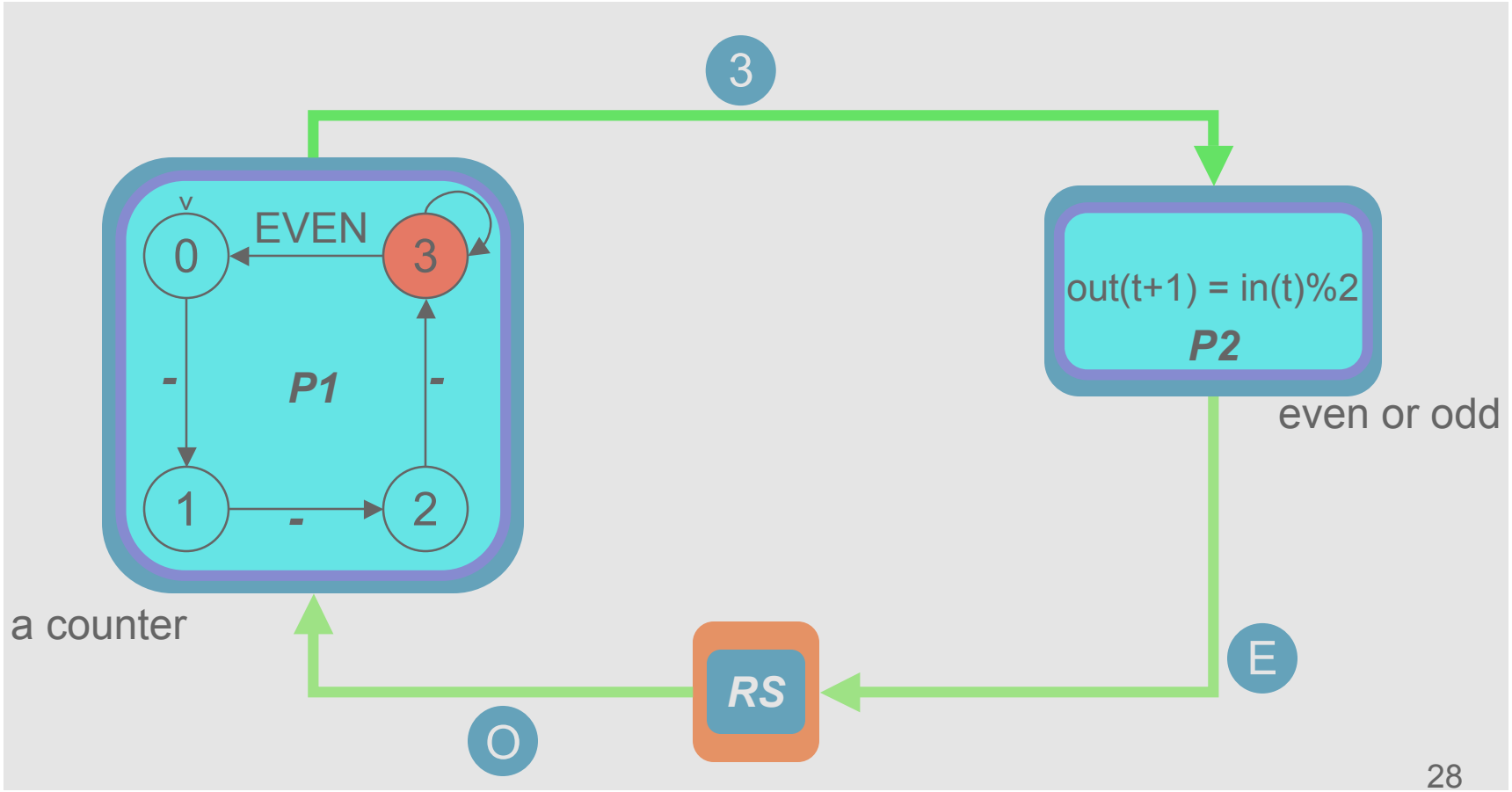
Throughput Optimization Using Functional Independence Conditions

- Roll back to cycle 5:
  - P2's trace: *OEOEO*
  - Reference: *OEOEOEOE…*



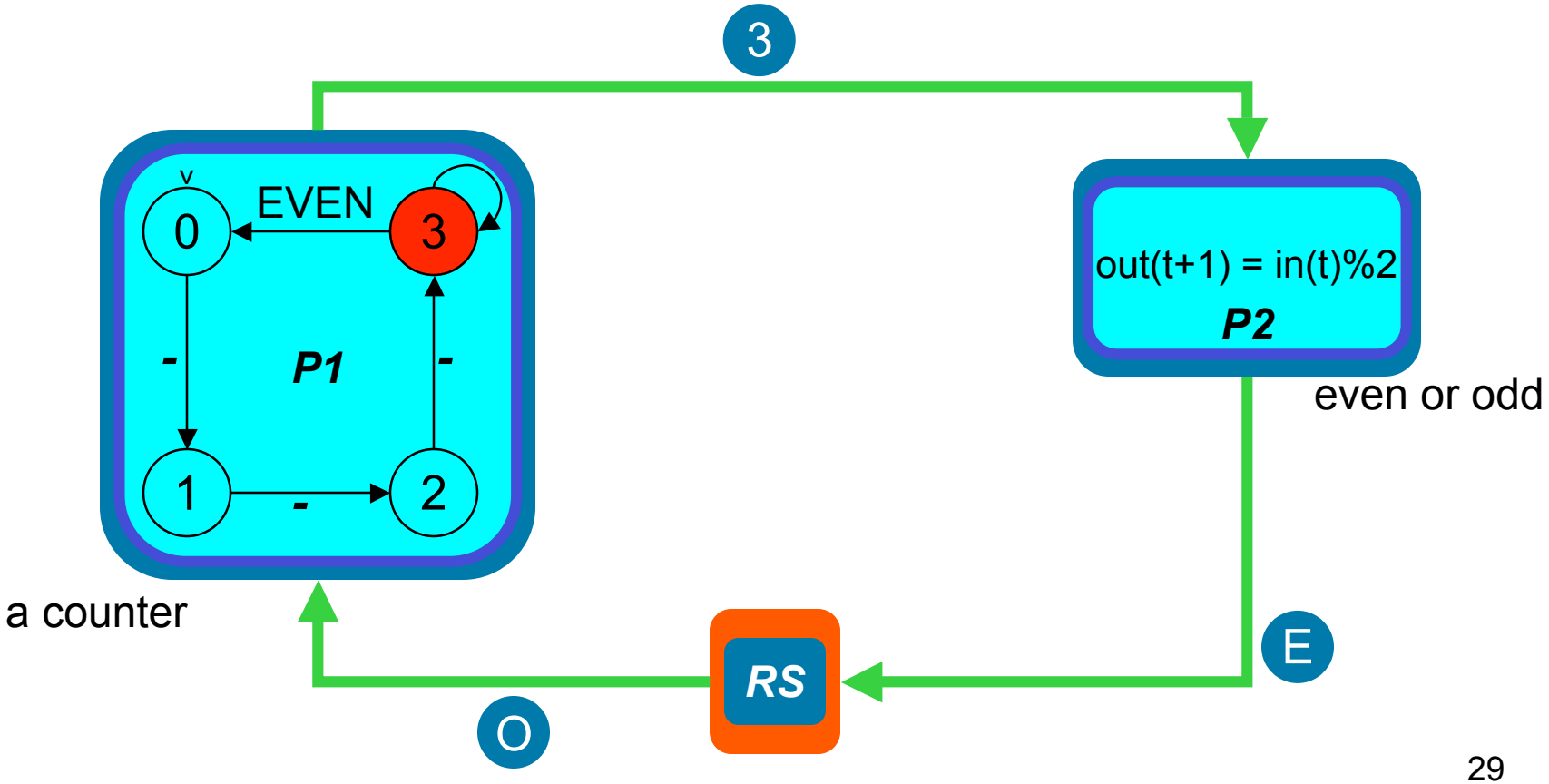
Throughput Optimization Using Functional Independence Conditions

- Roll back to cycle 4:
  - P2's trace: *OEOE*
  - Reference: *OEOEOEOE…*



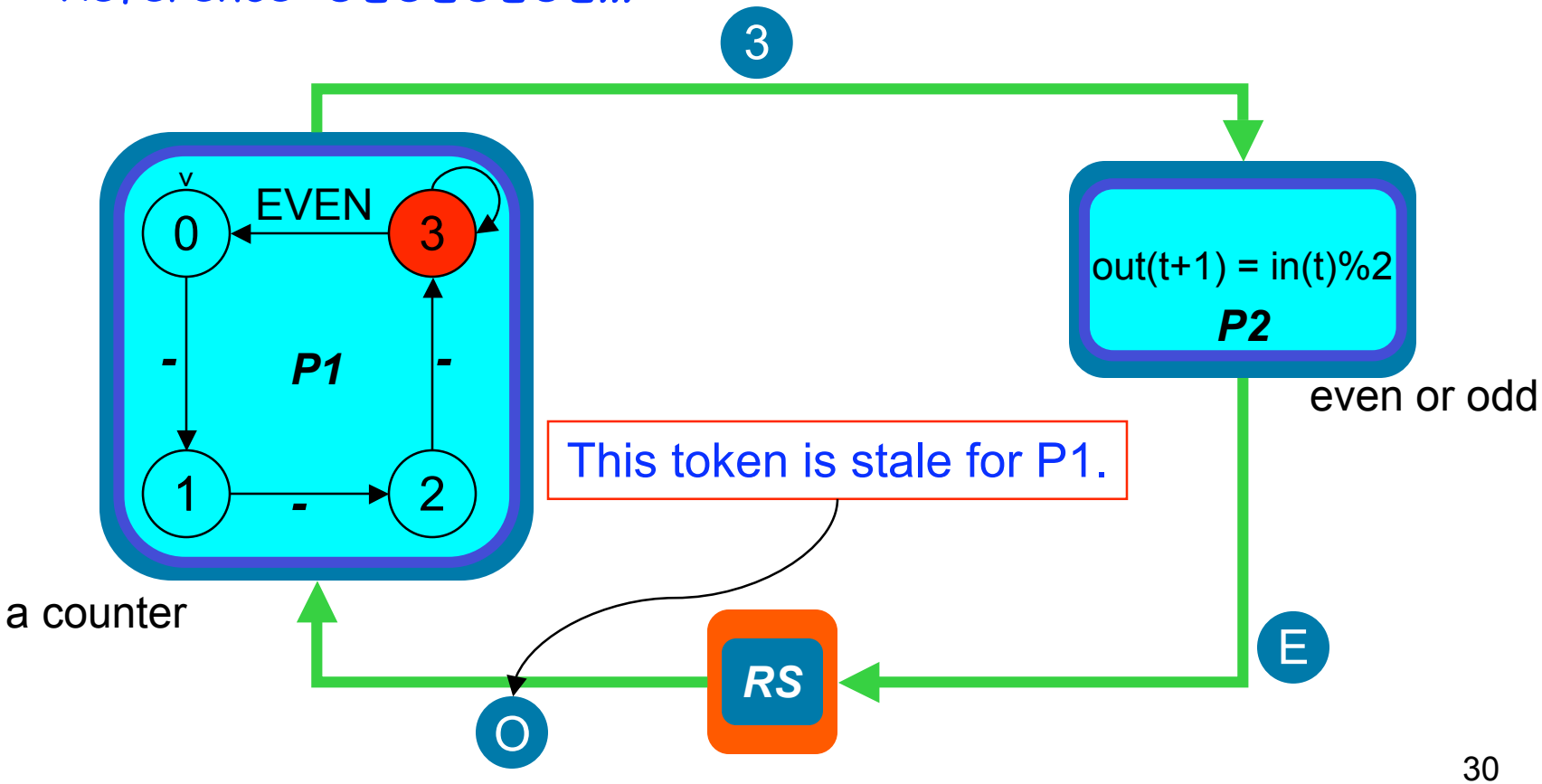
Throughput Optimization Using Functional Independence Conditions

- To see how to fix this, let's restart at cycle 4:
  - P2's trace: *OEOE*
  - Reference: *OEOEOEOE…*



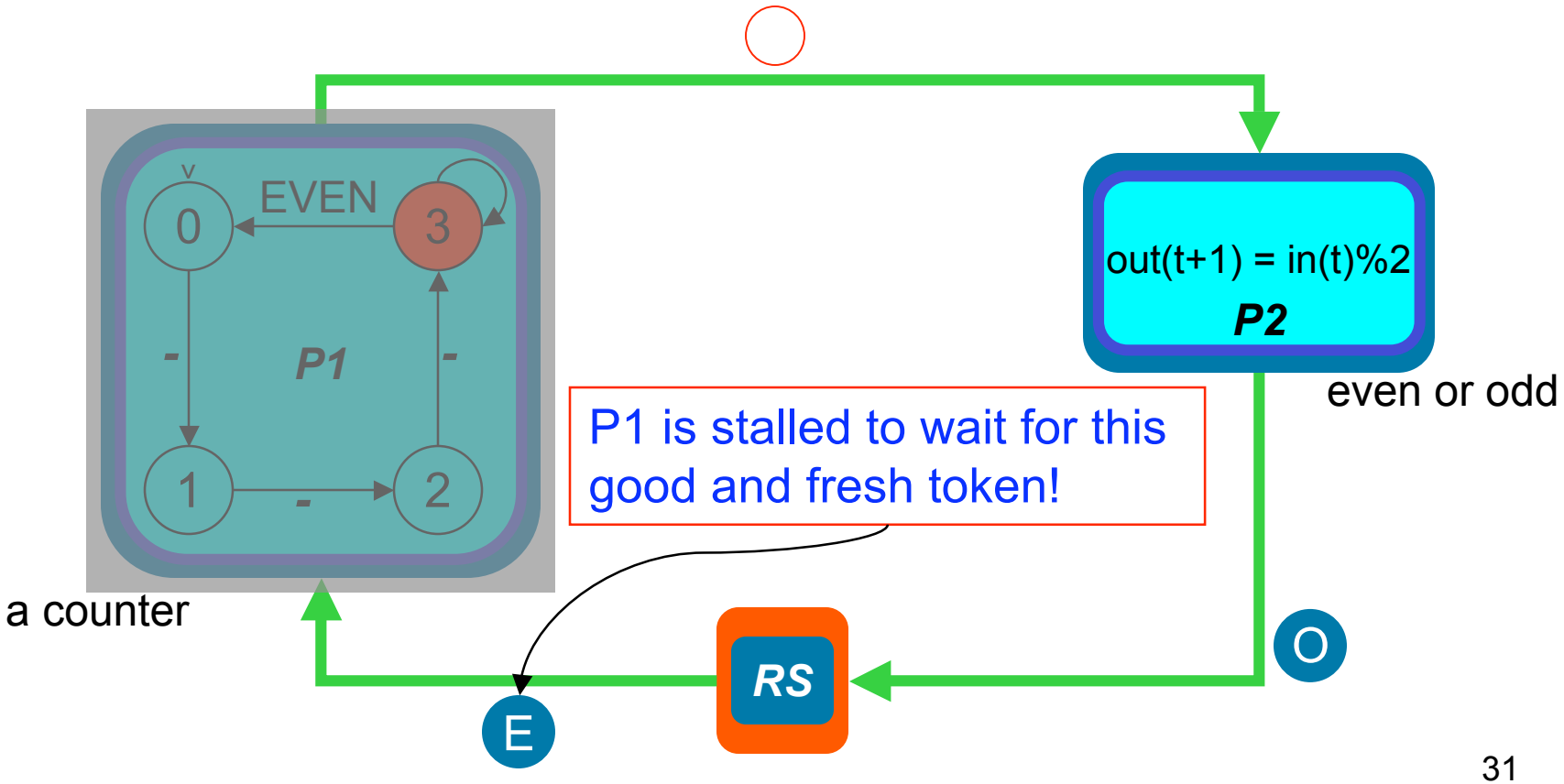
Throughput Optimization Using Functional Independence Conditions

- Cycle 4: P1's input "ODD" is valid but *stale*. NOT usable!
  - P1 should have been at state 2 to use it, not at state 3
  - P2's trace: *OEOE*
  - Reference: *OEOEOEOE…*



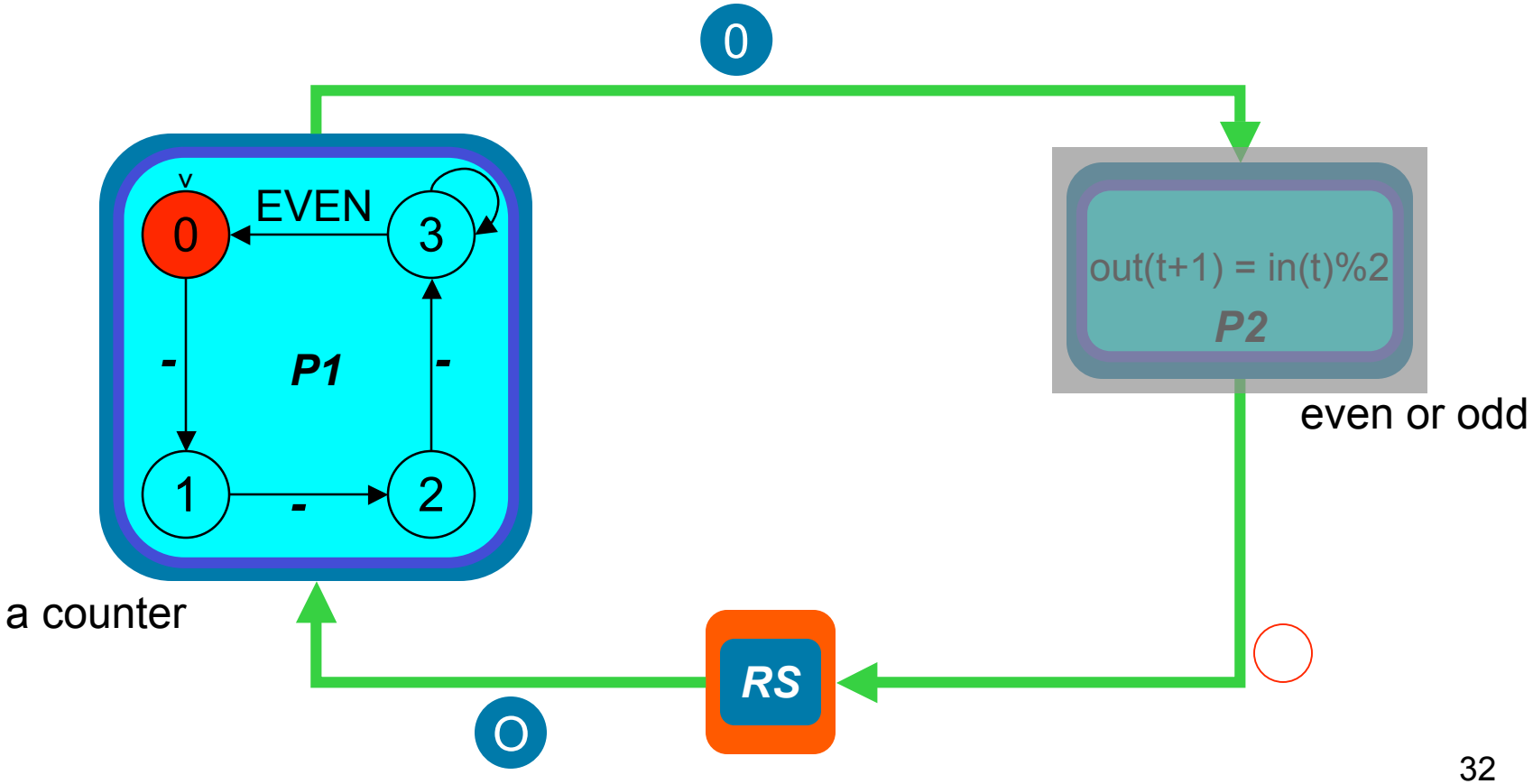
Throughput Optimization Using Functional Independence Conditions

- At cycle 5, P1 is stalled. Called "deferred stall"
  - P2's trace: *OEOEO*
  - Reference: *OEOEOEOE…*



Throughput Optimization Using Functional Independence Conditions

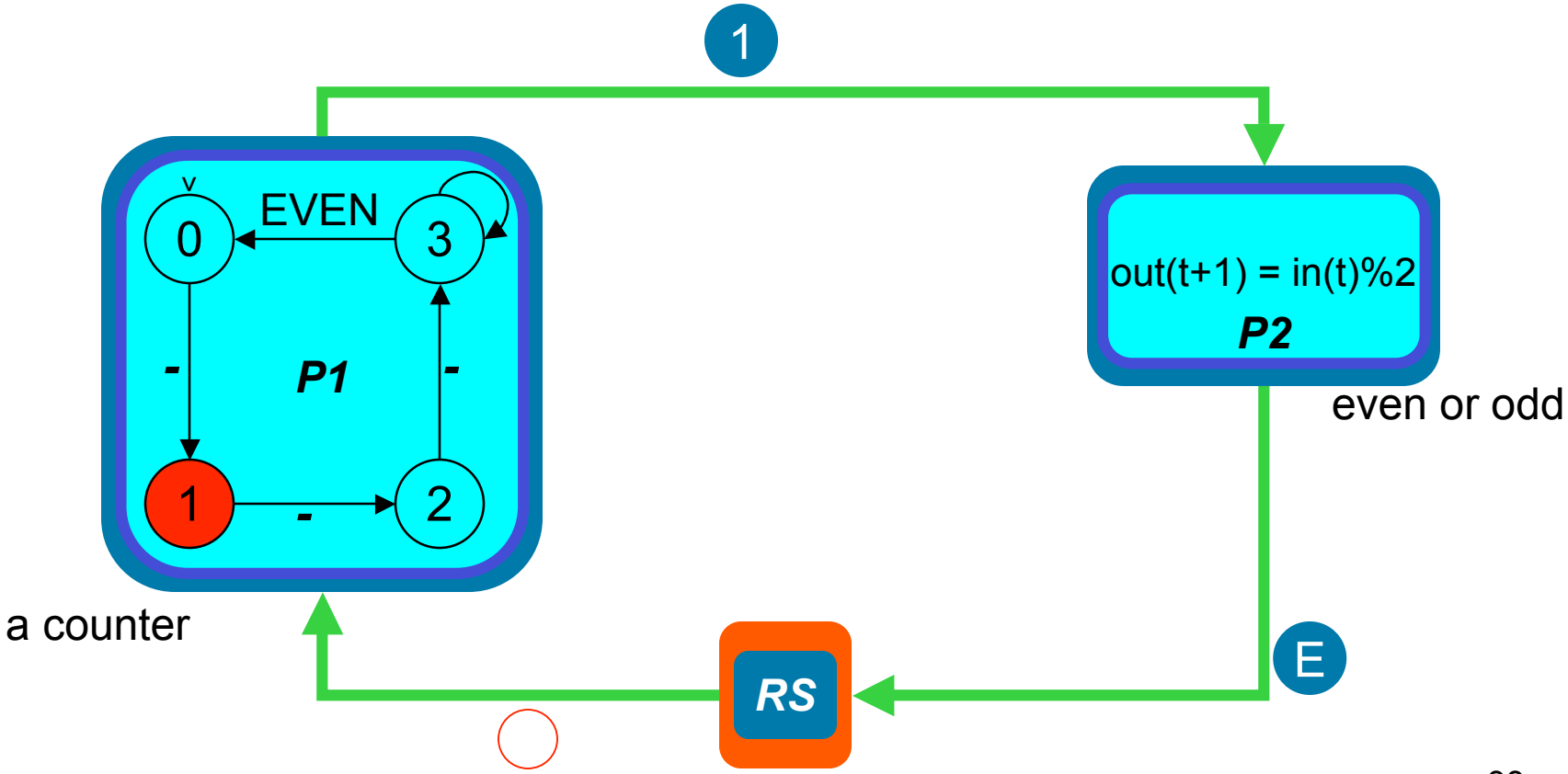
- Cycle 6:
  - P2's trace: *OEOEO#*
  - Reference: *OEOEOEOE…*



Throughput Optimization Using Functional Independence Conditions

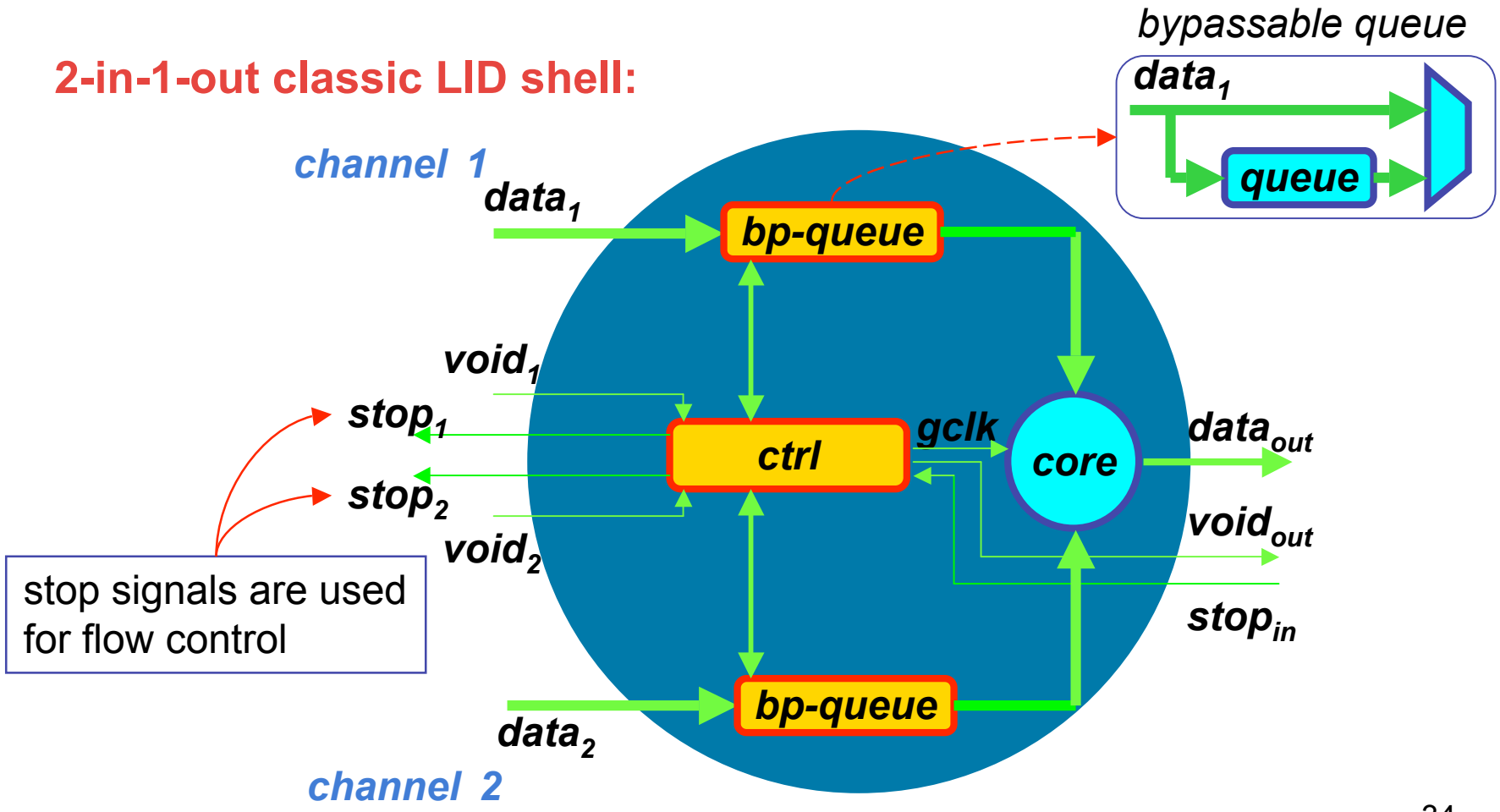
- Cycle 7 8 9…:
  - P2's trace: *OEOEO#EOEO# (correct!!)*
  - Reference: *OEOEOEOE…*

Throughput is increased to 4/5 (improved from 2/3)



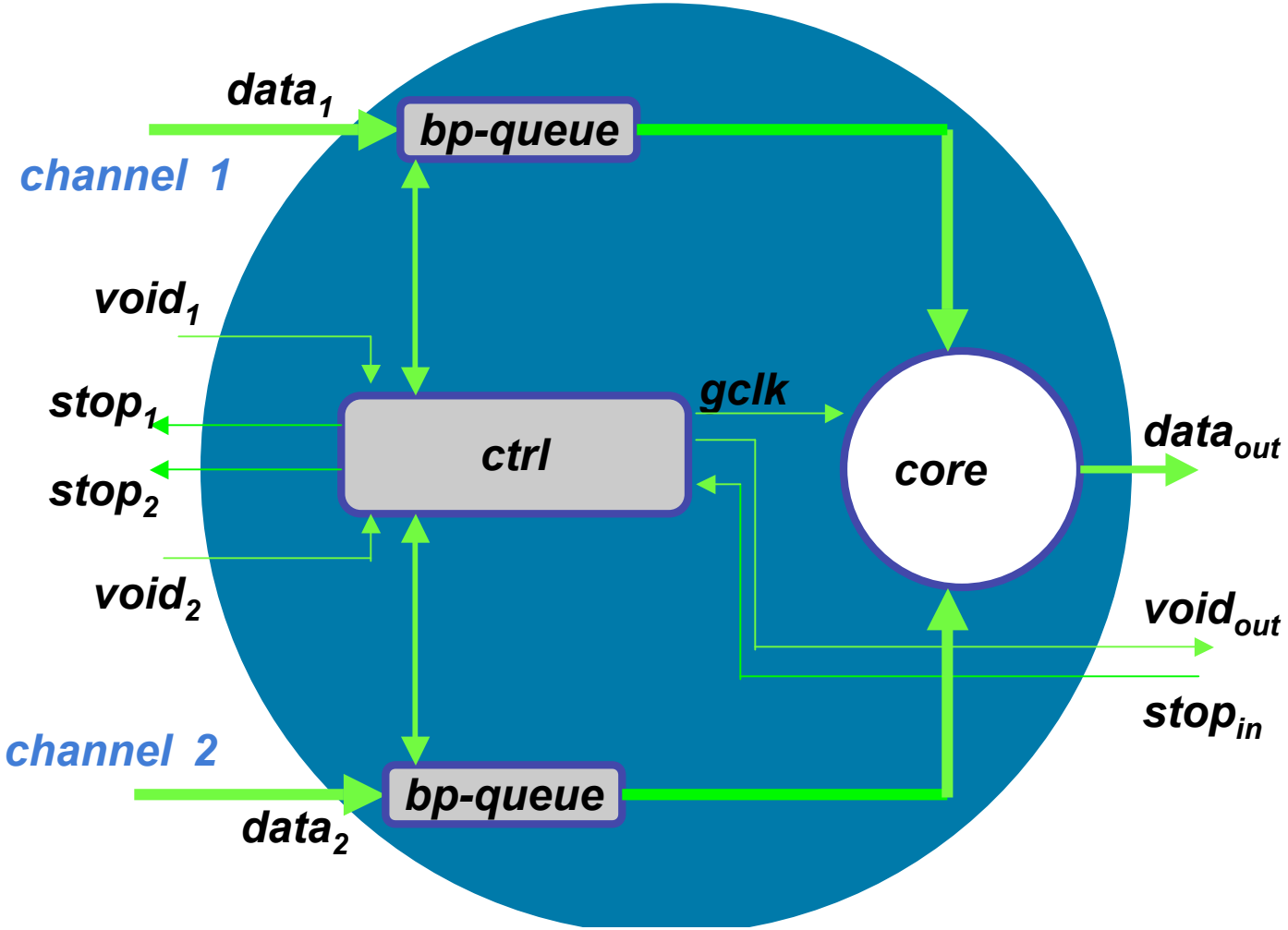
FIC-Shell Design

- FIC-shell is an extension to the classic LID shell



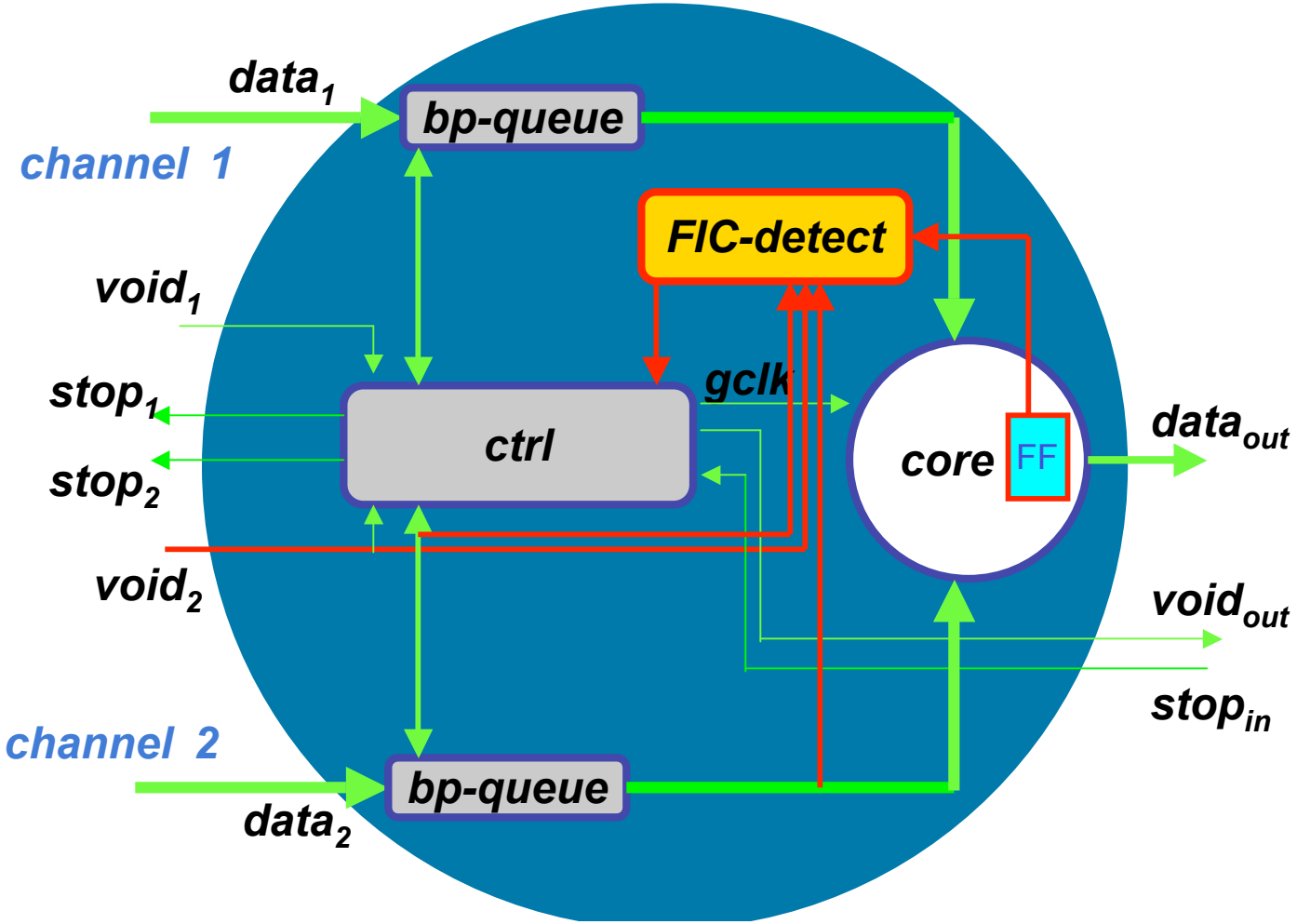
FIC-Shell Design

- In FIC-shell, each input channel also has:



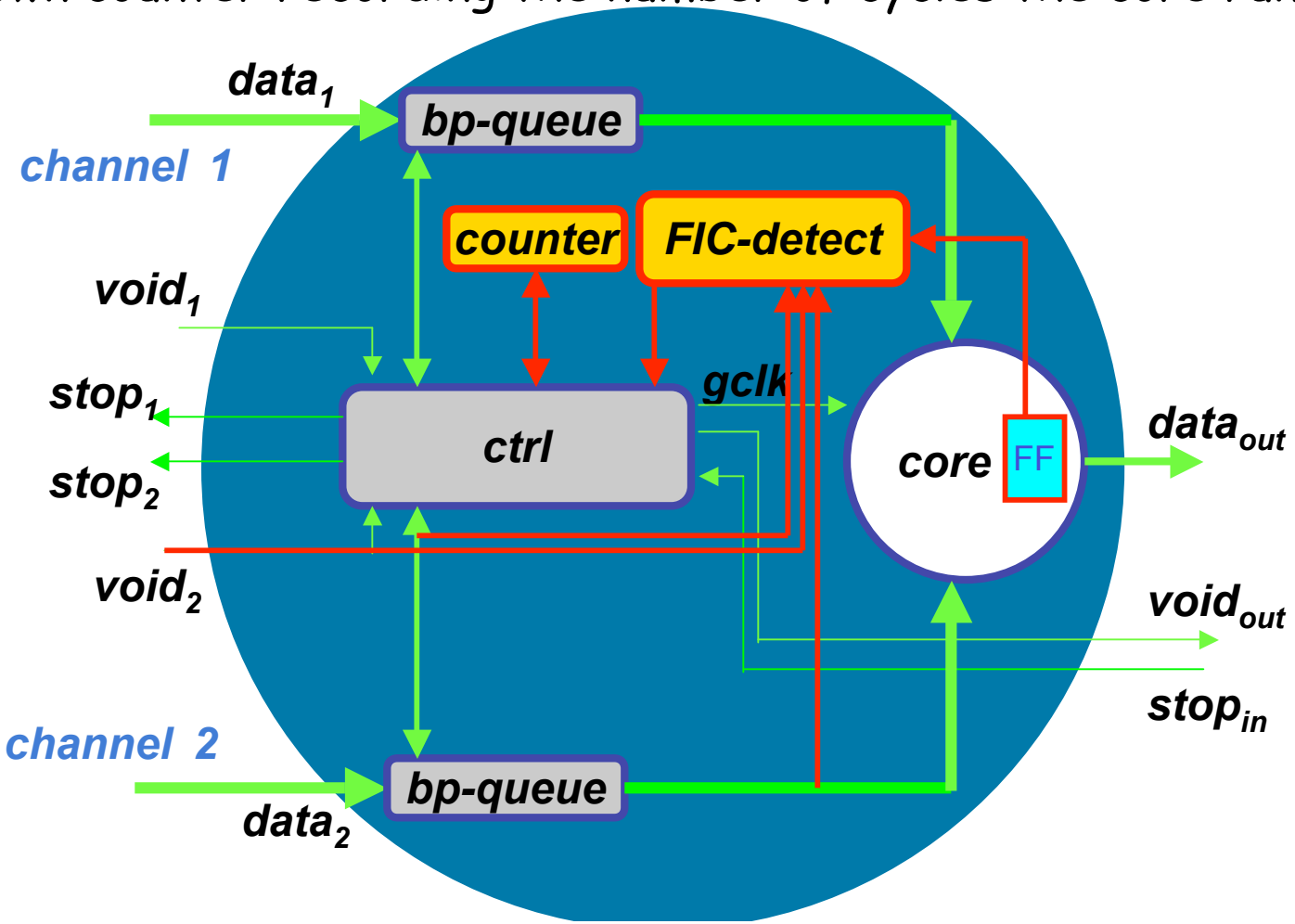
FIC-Shell Design

- In FIC-shell, each input channel also has:
  - FIC-detection logic (synthesized) telling when a FIC happens



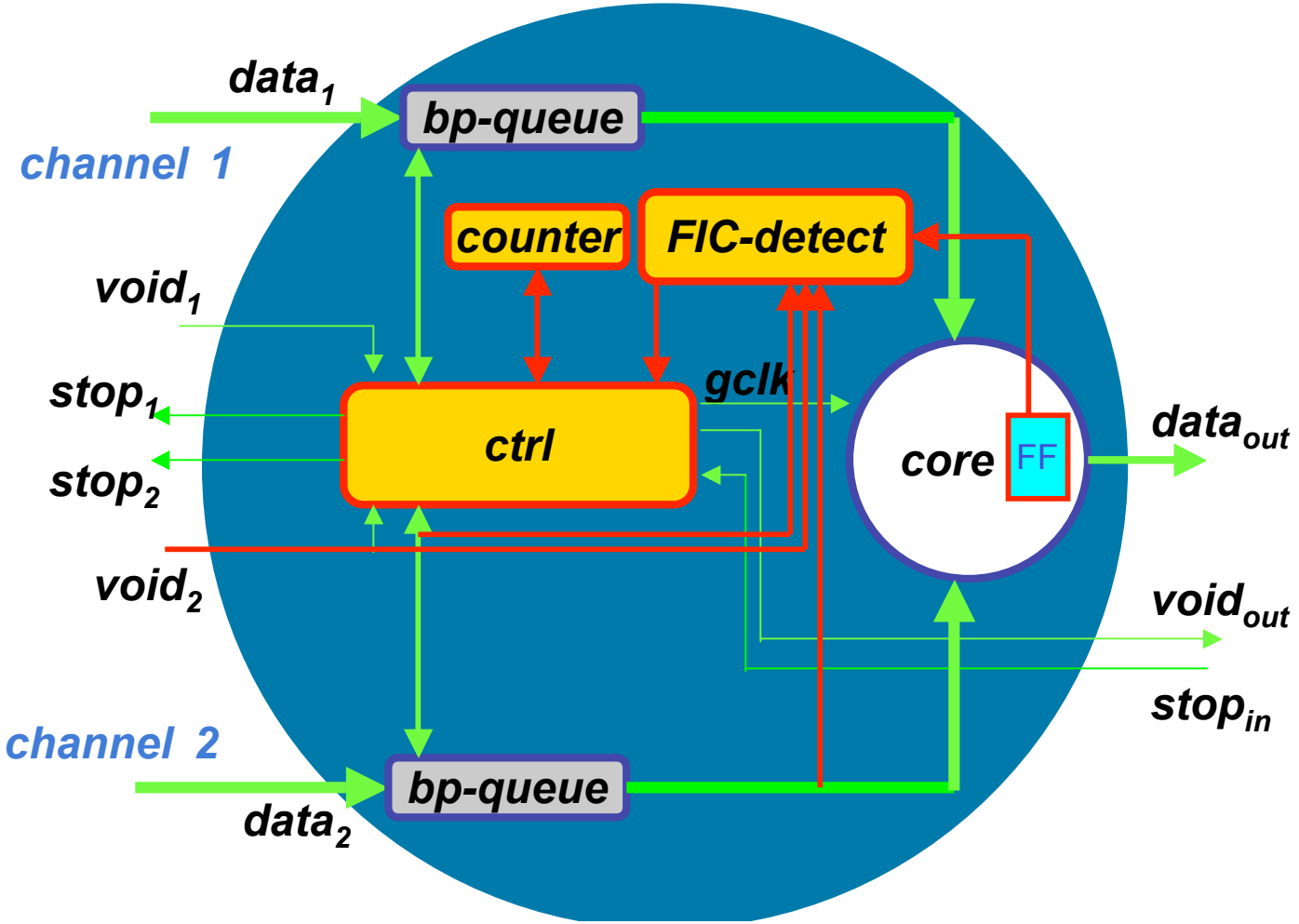
FIC-Shell Design

- In FIC-shell, each input channel also has:
  - FIC-detection logic (synthesized) telling when a FIC happens
  - up-down counter recording the number of cycles the core runs ahead



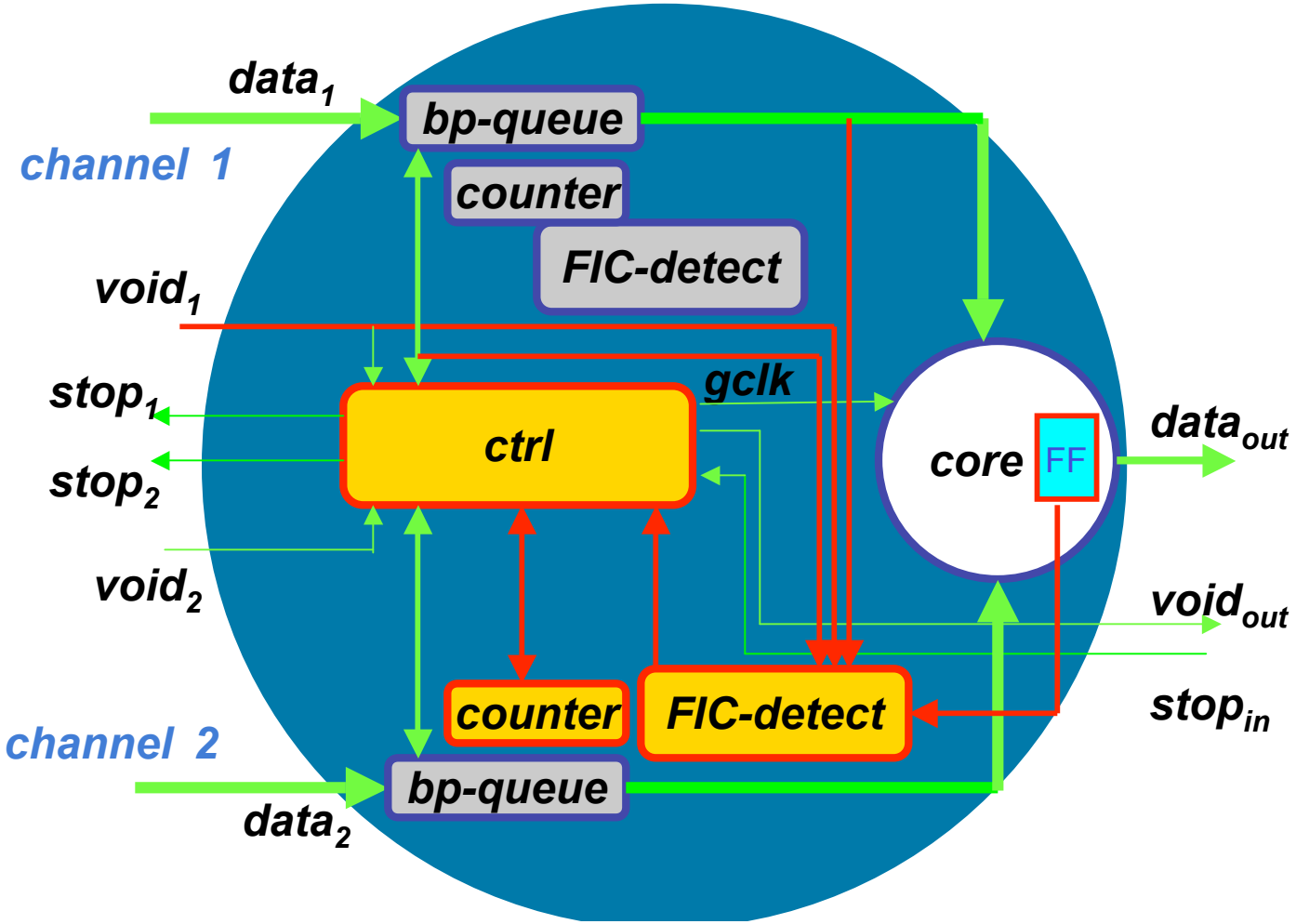
FIC-Shell Design

- Counter bookkeeping rules (roughly):
  - up if core is fired but no valid token is present
  - down if a valid token arrives but the core is stalled



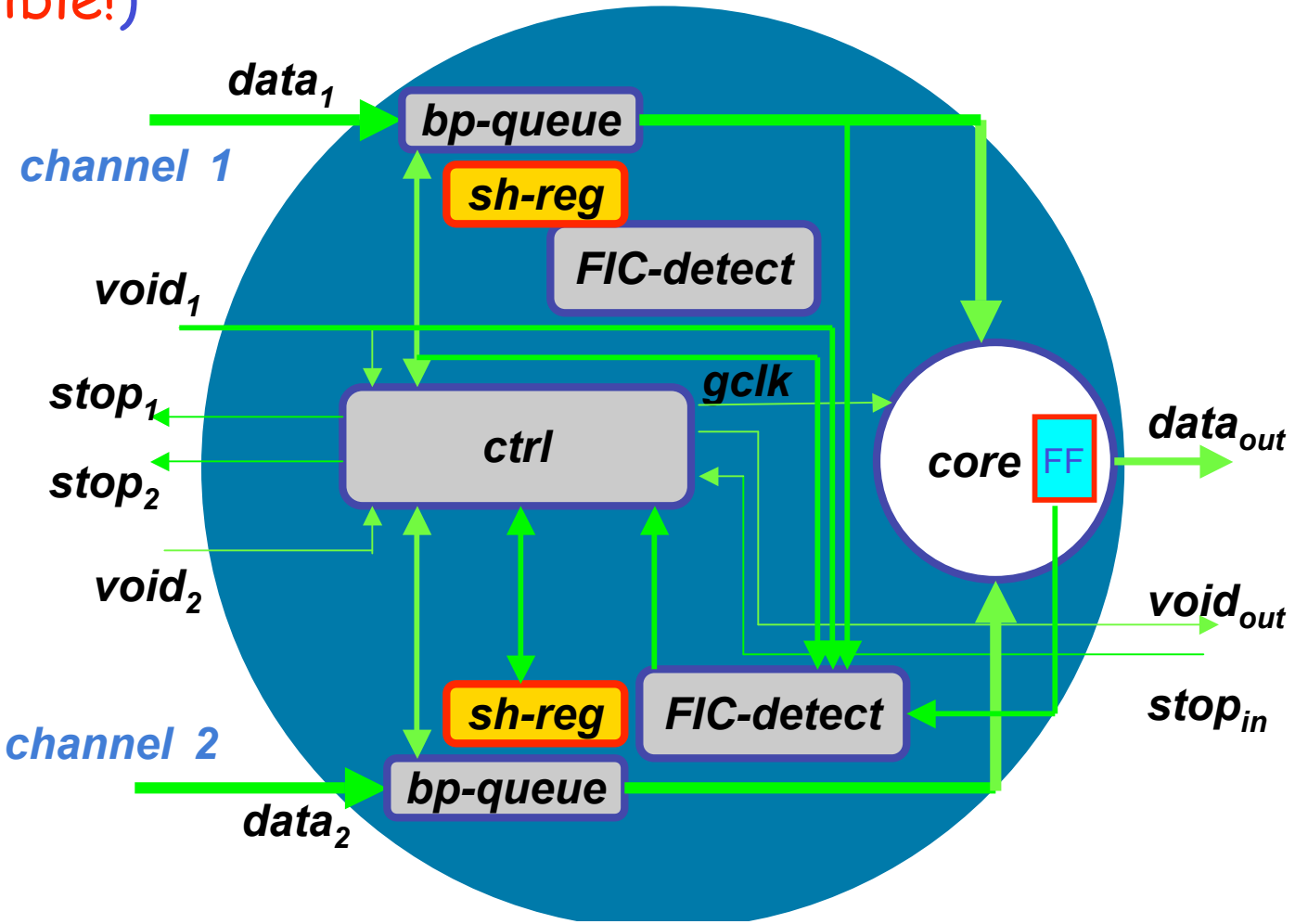
FIC-Shell Design

- Each channel has its own counter and FIC-detect



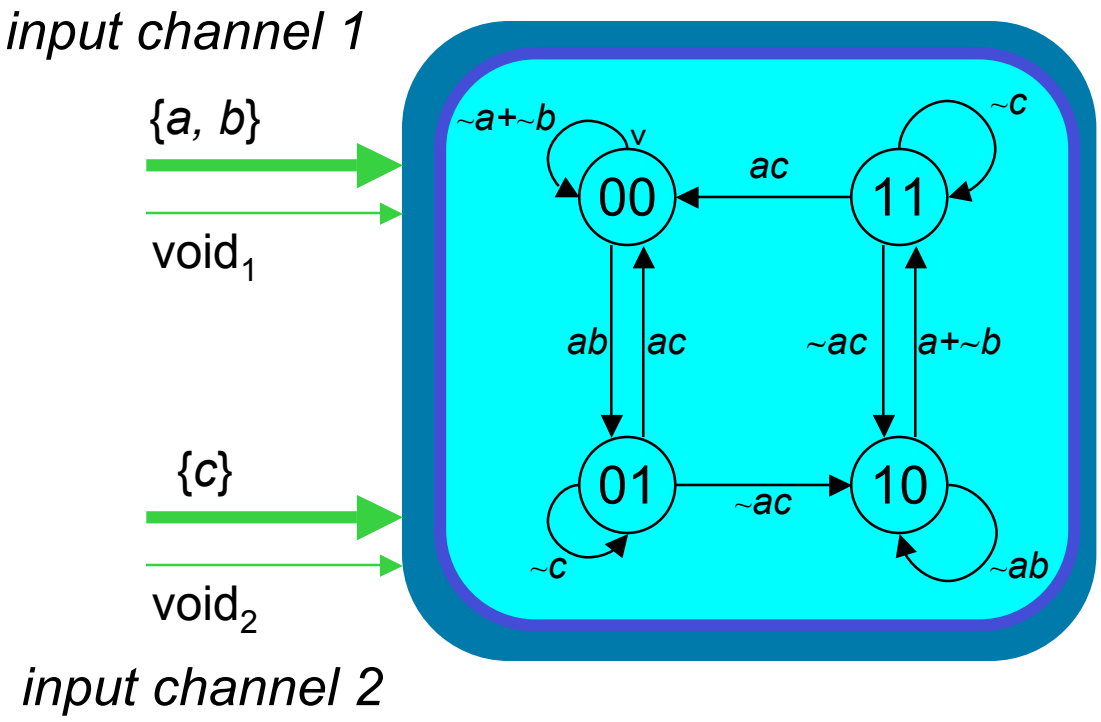
FIC-Shell Design

- The up-down counter can be replaced by a 1-bit wide shift register
- FIC-shells use the same protocol as classic shells (backward compatible!)



FIC-Detect Logic Synthesis Algorithm

Ex: a simple Moore FSM



4 states: $\{s_0, s_1\} = \{00, 01, 10, 11\}$

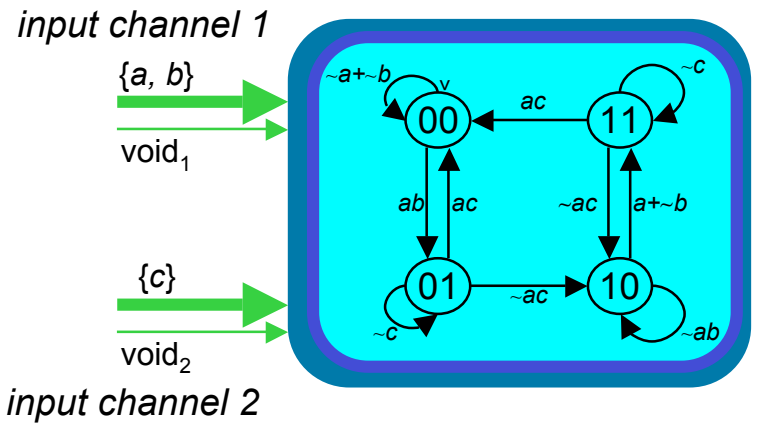
state transition functions:

$s_0' = s_1 \sim ac + s_0 \sim c + s_0 \sim s_1$

$s_1' = \sim s_1 ab + s_0 \sim s_1 \sim b + s_1 c$

FIC-Detect Logic Synthesis Algorithm

- Compute observability don't cares (ODCs) of state transition functions on all input variables



ODCs:

$ODC_a(s_0) = \sim s_1 + \sim c$

$ODC_a(s_1) = s_1 + \sim b$

$ODC_b(s_0) = 1$

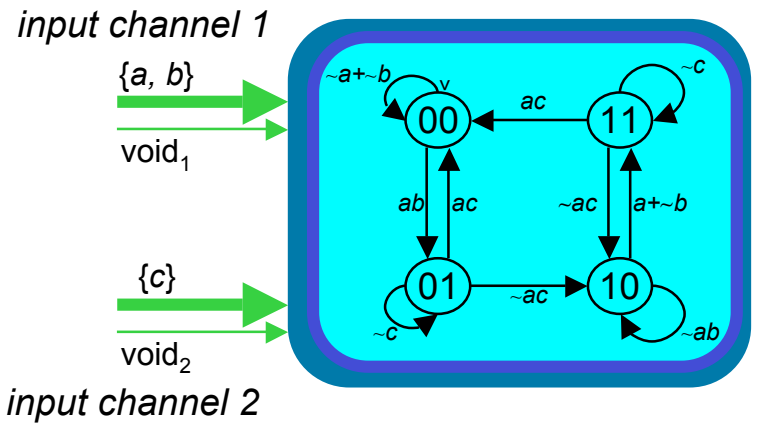
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$

$ODC_c(s_1) = \sim s_1$

FIC-Detect Logic Synthesis Algorithm

- Compute observability don't cares (ODCs) of state transition functions on all input variables



ODCs:

$ODC_a(s_0) = \sim s_1 + \sim c$

$ODC_a(s_1) = s_1 + \sim b$

$ODC_b(s_0) = 1$

$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$

$ODC_c(s_1) = \sim s_1$

If true, input c's value will not affect S_1's next value.

FIC-Detect Logic Synthesis Algorithm

- Take conjunctions of ODCs of the same input variables

$ODC_a(s_0) = \sim s_1 + \sim c$
$ODC_a(s_1) = s_1 + \sim b$

$\cap$

$ODC_b(s_0) = 1$
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$
$ODC_c(s_1) = \sim s_1$

$\cap$

FIC-Detect Logic Synthesis Algorithm

- Take conjunctions of ODCs on the same input variable

$ODC_a(s_0) = \sim s_1 + \sim c$
$ODC_a(s_1) = s_1 + \sim b$
$\cap \rightarrow ODC_a(S) = \sim s_1 \sim b + \sim c\, s_1 + \sim b \sim c$

$ODC_b(s_0) = 1$
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$
$\cap \rightarrow ODC_b(S) = s_1 + s_0 + \sim(s_0 a)$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$
$ODC_c(s_1) = \sim s_1$
$\cap \rightarrow ODC_c(S) = \sim s_1 + s_0 \sim s_1 a + \sim s_0 \sim s_1 a$

FIC-Detect Logic Synthesis Algorithm

- Take conjunctions (and consensus) on variables of the same channel

$ODC_a(s_0) = \sim s_1 + \sim c$
$ODC_a(s_1) = s_1 + \sim b$

$\cap$

$ODC_b(s_0) = 1$
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$
$ODC_c(s_1) = \sim s_1$

input variables of channel 1

$ODC_a(S) = \sim s_1 \sim b + \sim c\, s_1 + \sim b \sim c$

$\cap$

$ODC_b(S) = s_1 + s_0 + \sim(s_0 a)$

input variables of channel 2

$ODC_c(S) = \sim s_1 + s_0 \sim s_1 a + \sim s_0 \sim s_1 a$

$\cap$

FIC-Detect Logic Synthesis Algorithm

- Take conjunctions on variables of the same channel

$ODC_a(s_0) = \sim s_1 + \sim c$
$ODC_a(s_1) = s_1 + \sim b$

$\cap$

$ODC_b(s_0) = 1$
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$
$ODC_c(s_1) = \sim s_1$

$\cap$

$ODC_a(S) = \sim s_1 \sim b + \sim c\, s_1 + \sim b \sim c$

$\cap$

$ODC_b(S) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

$ODC_c(S) = \sim s_1 + s_0 \sim s_1 a + \sim s_0 \sim s_1 a$

$\cap$

$FIC_{in1}(S) = s_1 \sim c \cap (c \text{ is valid})$

$FIC_{in2}(S) = \sim s_1$

FIC-Detect Logic Synthesis Algorithm

$ODC_a(s_0) = \sim s_1 + \sim c$
$ODC_a(s_1) = s_1 + \sim b$

$\cap$

$ODC_b(s_0) = 1$
$ODC_b(s_1) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

$ODC_c(s_0) = \sim s_1 + s_0 \sim a + \sim s_0 a$
$ODC_c(s_1) = \sim s_1$

$\cap$

$ODC_a(S) = \sim s_1 \sim b + \sim c\, s_1 + \sim b \sim c$

$\cap$

$ODC_b(S) = s_1 + s_0 + \sim(s_0 a)$

$\cap$

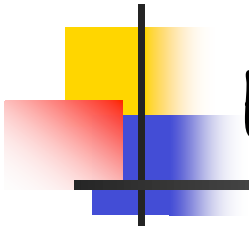
$ODC_c(S) = \sim s_1 + s_0 \sim s_1 a + \sim s_0 \sim s_1 a$

$\cap$

The final FICs:

$$FIC_{in1}(S) = s_1 \sim c \cap (c \text{ is valid})$$

$$FIC_{in2}(S) = \sim s_1$$



Experimental Results

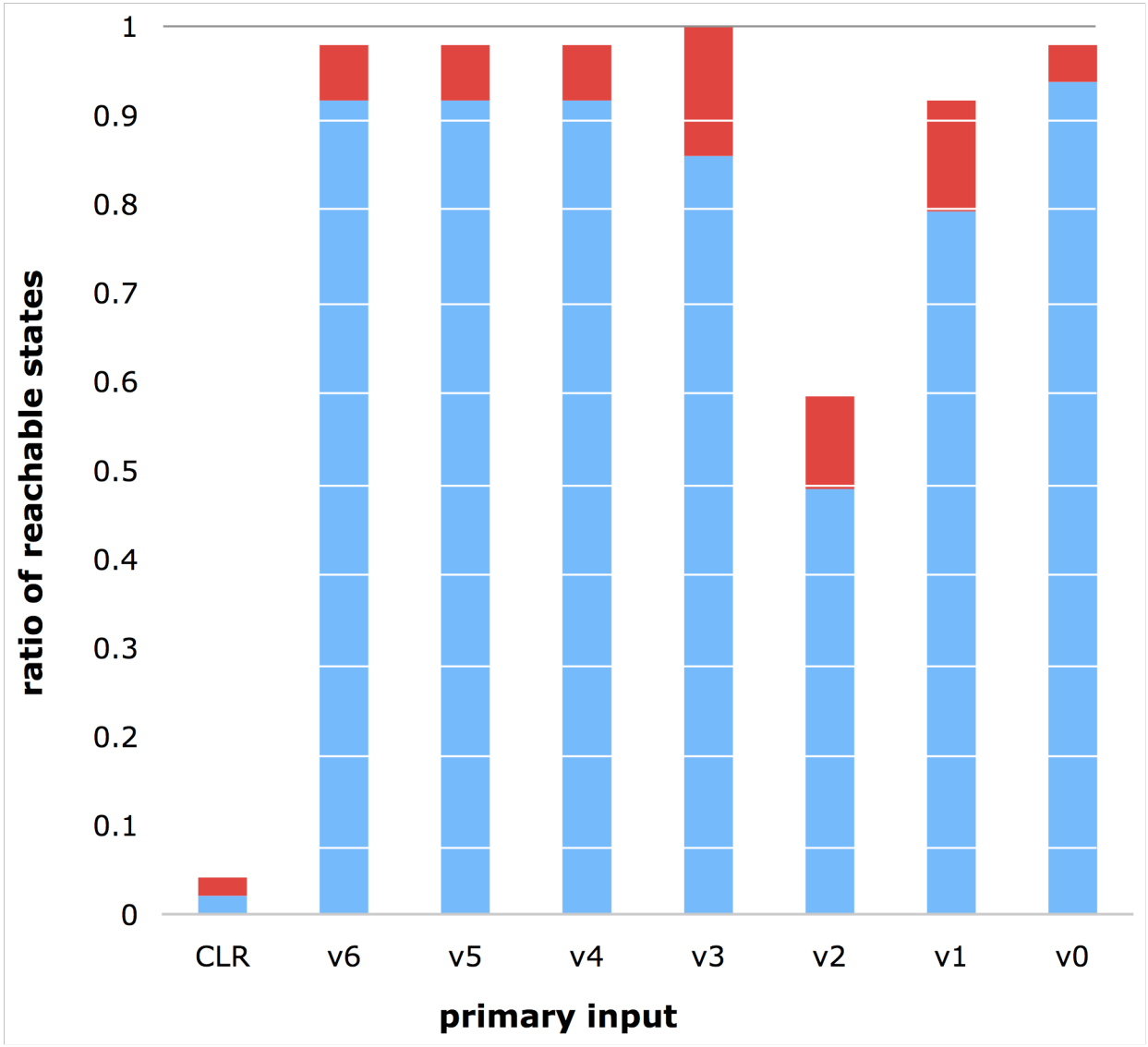
- Q1: Are real designs amenable to FIC optimization?
  - In ISCAS-89 and other seq. circuits:
    - 94% of reachable states have FIC in average
    - 72% of reachable states have some inputs not needed
      - faster & smaller FIC-detect logic !!

ISD-FIC: FIC depending on inputs and states

SD-FIC : FIC depending only on states

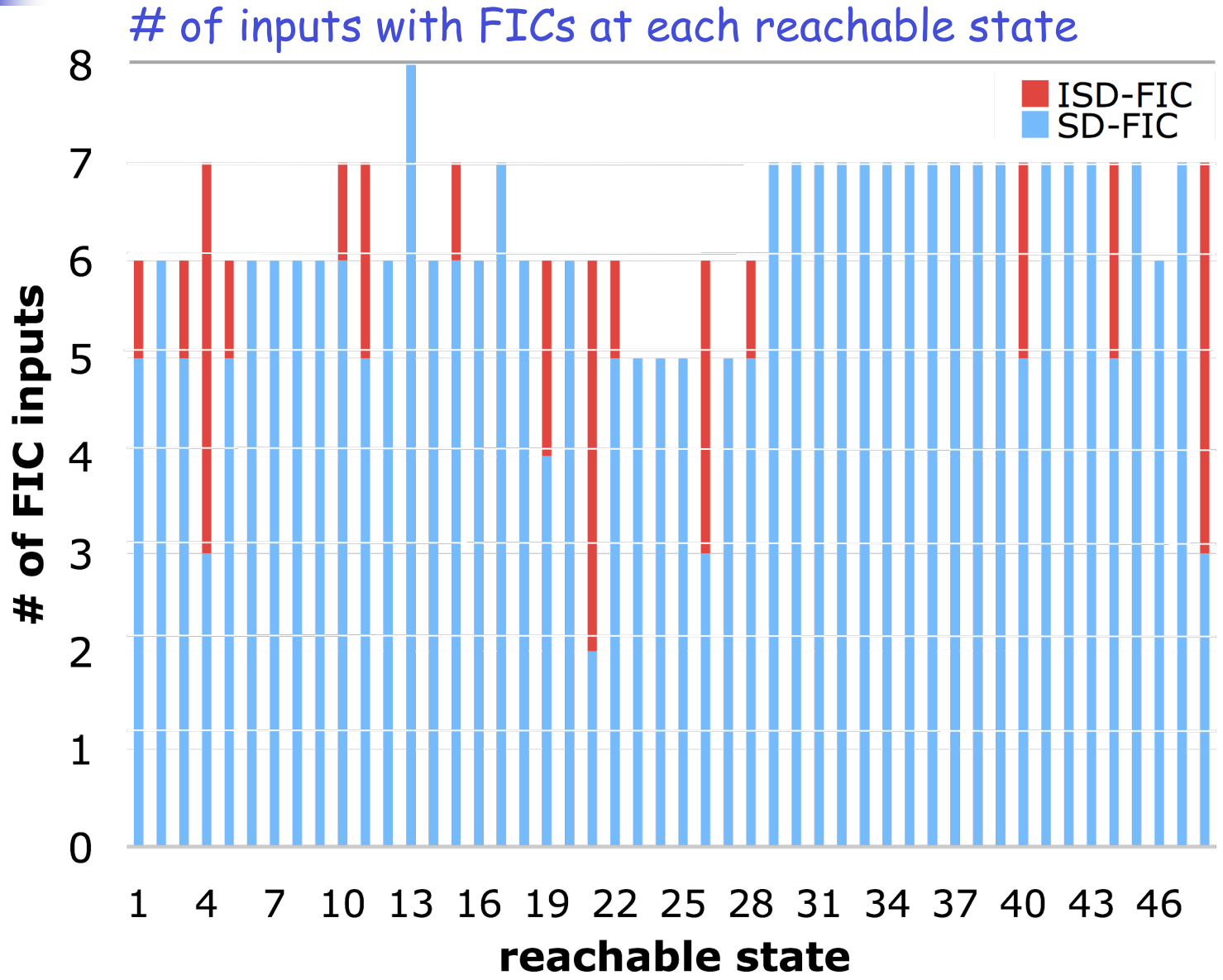
Experimental Results: s1488

Ratio of states having FICs



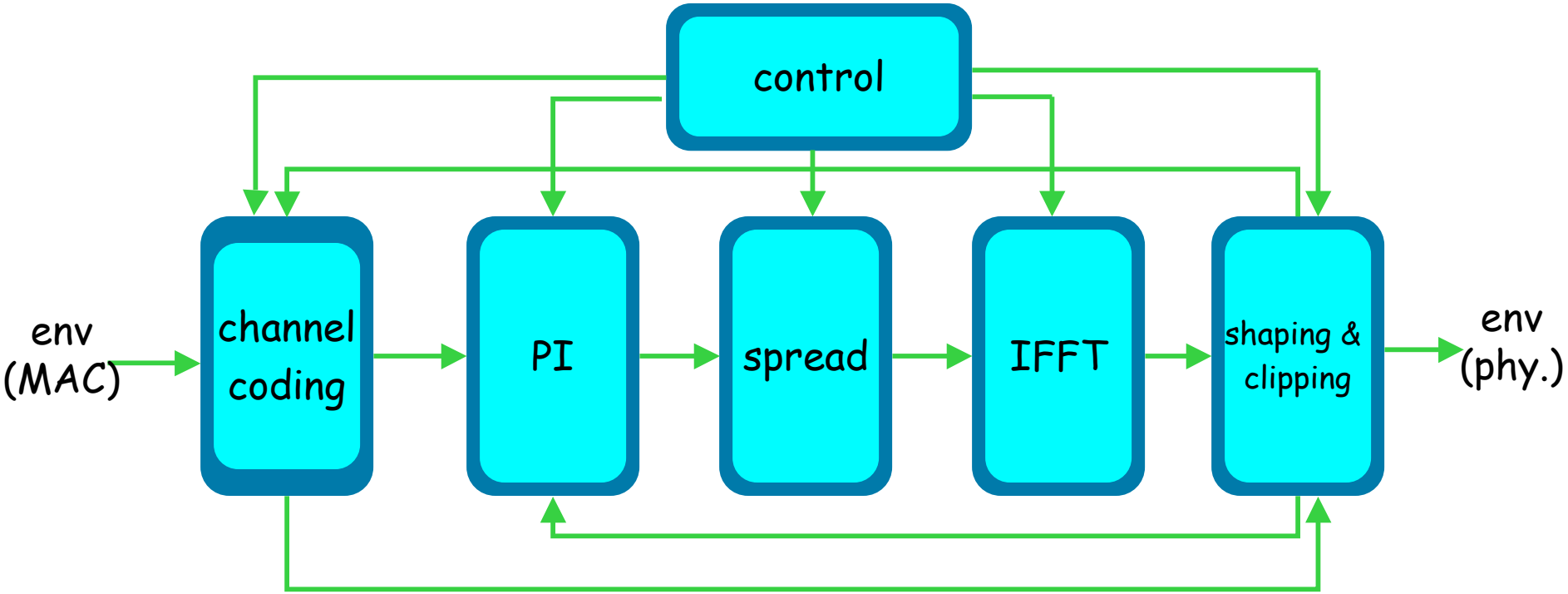
ISD-FIC
SD-FIC
(input not needed)

Experimental Results: s1488

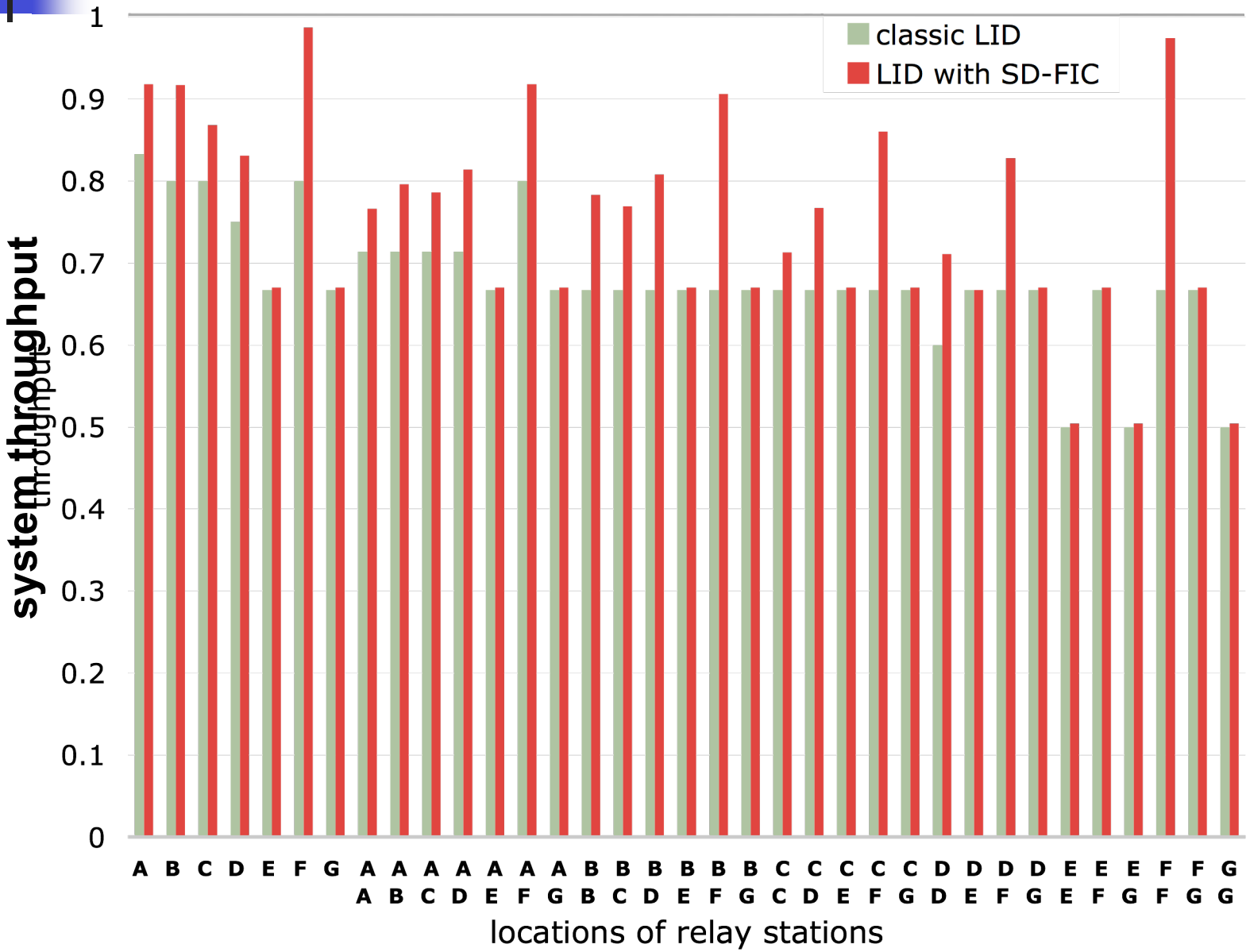


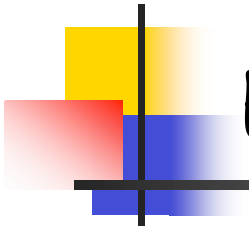
Experimental Results: COFDM

- Q2: Does FIC increase system throughput on real designs?
  - By 10% in average on a real design (coded orthogonal frequency division modulator, COFDM, [Liu *et. al.* ISSCC'05])



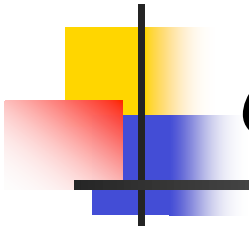
Experimental Results: COFDM





Experimental Results

- Q3: What's the area/delay overhead of FIC shells?
  - On COFDM, classic shells account for 1%~3% area overhead, depending on sizes of the input queues (synthesized & mapped to standard cells in 90nm)
  - FIC optimization adds negligible (< 0.01%) area to classic shells
  - Classic/FIC-shells has no impact on critical path delays (based on static timing analysis)
    - FIC optimization increases the processing data rate (data/sec.) of the system



Conclusion

- Propose a novel shell design exploiting FIC to increase system performance for LI systems
- A fully automatic algorithm to synthesize FIC-detection logic
- 1st empirical study of LID on a real design
- FIC optimization increases throughput by 10% with little area overhead on the real circuit

Thank you!